UNIVERSIDADE DE SÃO PAULO
FACULDADE DE ECONOMIA, ADMINISTRAÇÃO E CONTABILIDADE
DEPARTAMENTO DE ADMINISTRAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ADMINISTRAÇÃO

UMA CONTRIBUIÇÃO AO ESTUDO DOS TESTES PARA A GARANTIA DE QUALIDADE DE *SOFTWARE*

Alexandre Moreira Nascimento

Orientador: Prof. Dr. Antônio César Amarú Maximiano

SÃO PAULO
2005

Prof. Dr. Adolpho Jose Melfi
Reitor da Universidade de São Paulo

Profa. Dra. Maria Tereza Leme Fleury
Diretora da Faculdade de Economia, Administração e Contabilidade

Prof. Dr. Eduardo Pinheiro Gondim de Vasconcellos
Chefe do Departamento de Administração

Prof. Dr. Isak Kruglianskas
Coordenador do Programa de Pós-Graduação em Administração

ALEXANDRE MOREIRA NASCIMENTO

UMA CONTRIBUIÇÃO AO ESTUDO DOS TESTES PARA A GARANTIA DE QUALIDADE DE *SOFTWARE*

Dissertação apresentada ao Departamento de Administração da Faculdade de Economia, Administração e Contabilidade da Universidade de São Paulo como requisito para a obtenção do título de Mestre em Administração.

Orientador: Prof. Dr. Antônio César Amarú Maximiano

SÃO PAULO

2005

Dissertação defendida e aprovada, em 30.09.2005, no Programa de Pós-Graduação em Administração, pela seguinte comissão julgadora:
Prof. Dr. Antonio César Amaru Maximiano
Profª Drª Isak Kruglianskas
Prof. Dr. José Reinaldo Silva

Dedico este trabalho aos meus pais,
irmãos e amigos.

Agradeço ao Prof. Dr. Antônio César Amarú Maximiano pela infindável paciência e ajuda. Agradeço também ao Eric Brandt Schonwald pelo suporte nas ferramentas de pesquisas, bem como ao meu irmão André pela ajuda nas revisões.

"Não há uma pergunta que resida em nós e uma resposta que esteja nas coisas, um ser exterior a descobrir e uma consciência observadora: a solução está também em nós e o próprio ser é problemático. Há algo da natureza da interrogação que se transfere para a resposta."

Maurice Merleau-Ponty

RESUMO

Este trabalho revisa os principais conceitos ligados à qualidade, bem como os testes para a garantia de qualidade dos *softwares*, e traz uma análise das práticas de teste utilizadas nas empresas de diversos portes e setores de atuação, do grau conhecimento dos atributos da qualidade de *software* por parte destas organizações, e, por fim, apresenta uma avaliação da qualidade dos processos de *software* empregados nestas empresas, comparando-os a uma organização de testes de classe mundial, de acordo com o modelo de avaliação proposto pelo QAI (Quality Assurance Institute).

ABSTRACT

This work revises the main quality's concepts, as well as the tests for software quality assurance; analysis the use of tests by companies and the degree of knowledge of software quality attributes; and finally, presents an evaluation of the software quality processes used by these companies, comparing them against a world-class testing organization, using the Quality Assurance Institute assessment model.

SUMÁRIO

LISTA DE ABREVIATURAS

CDR: Critical Design Review
CM: Change Management
CMM: Capability Maturity Model
COA: Cost of Achieving Quality
COF: Cost of Failure
COQ: Cost of Quality
CRM: Customer Relationship Management
ETVX: Entry-Task-Validation-Exit
FA: Functional Audit
FP: Function Point
GUI: Graphical User Interface
HLD: High Level Design
IDP: Iterative Development Process
IE: Iterative Enhancement
IEEE: Institute of Electrical & Electronic Engineers
ISO: International Standards Organization
LCL: Lower Control Level
LLD: Low Level Design
LOC: Line of Code
MTBF: Mean Time Between Failure
OO: Object Oriented
PA: Physical Audit
PDR: Preliminary Design Review
PIR: Post Implementation Review
QAI: Quality Assurance Institute
RUP: Rational Unified Process
SEI: Software Engeneering Institute
SLC: Software Life Cycle
SQA: Software Quality Assurance
SQS: Software Quality System
SRR: Software Requirements Review
STEP: Systematic Test and Evolution Process
TRR: Test Readiness Review
UCL: Upper Control Level
UML: Universal Modeling Language
UT: Unit Test
XP: Extreme Programing
YANGI: You aren't gonna need it

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE ILUSTRAÇÕES

1 CONTEXTUALIZAÇÃO DA PESQUISA

1.1 Introdução

Este capítulo busca contextualizar a pesquisa realizada no escopo deste trabalho, apresentando inicialmente as justificativas da relevância e da importância do tema estudado. Para isto são descritos alguns fatos onde falhas em *softwares* ocasionaram impactos e prejuízos significantes. Em seguida, a pergunta de pesquisa, bem como os objetivos do estudo são descritos.

1.2 Justificativas do Estudo

No mundo moderno, o *software* passou a ter importante papel no apoio aos negócios das empresas. Esta importância tende a crescer em um mundo em que as atividades e produtos tendem a depender cada vez mais do *software*. Desde os primeiros computadores comerciais, os *softwares* implantados ou lançados no mercado têm se caracterizado, na sua maioria, por apresentarem um grande número de defeitos que afetam a usabilidade, a funcionalidade, a segurança e a confiabilidade dos mesmos, com impactos decisivos nos negócios, resultando, muitas vezes, em enormes prejuízos pela perda de participação de mercado ou por danos na imagem dos produtos lançados (FILHO e RIOS, 2003 apud COSTA, 2003, p.12).

Broekman e Notenboom (2003, p. x) também afirmam que a importância do *software* está crescendo muito em praticamente todos os setores da indústria, e, que na administração de empresas e negócios, o *software* se tornou uma tecnologia fundamental e indispensável. Os autores descrevem ainda que cada vez mais os produtos e inovações se baseiam em *software*, como no exemplo da indústria automotiva, onde as inovações baseadas em *software* para aumento da segurança, conforto e redução de consumo e emissão de poluentes é cada vez maior. Nos carros modernos de classe superior e de luxo, entre 20 a 25% do custo é devido à eletrônica e *software* embarcado, e, estima-se que esta proporção crescerá até 40% nos próximos 10 anos. (BROEKMAN; NOTENBOOM, 2003, p. x)

O *software* ainda tem uma participação importante na qualidade dos produtos e na produtividade de uma empresa. No entanto, infelizmente, a prática demonstra que é impossível desenvolver um sistema complexo baseado em *software* correto na primeira vez. ("first-time-right") (BROEKMAN; NOTENBOOM, 2003, p. x).

Atualmente, as vendas totais de *software* atingiram US$180 bilhões por ano nos Estados Unidos. Por outro lado, os *bugs* em *softwares* têm causado catástrofes de tempos em tempos (LI; WU, 2004, p. 2).

Em abril de 1999, um defeito de *software* provocou a falha no lançamento de um satélite militar de $1.2 bilhões no Cabo Canaveral. Esta é talvez a falha de *software* mais cara na história da indústria de *software*. Este fato disparou uma revisão completa nos programas de lançamento espaciais americanos na indústria e no meio militar, incluindo os processos de integração e teste de *software* (LI; WU, 2004, p. 2).

Outro problema causado por *software* em 1999 foi a perda do NASA Mars Climate Orbiter no espaço. Isto ocorreu por uma falha em um trecho do *software* que convertia as unidades de medida do sistema inglês para o sistema métrico (LI; WU, 2004, p. 2).

Outros prejuízos graves devido erros em *software* descritos por McConnell (2003, p. 1):

- problemas no sistema de controle de bagagens causou um atraso de mais de um ano na inauguração do aeroporto internacional de Denver. O custo diário do atraso foi estimado em US$1,1 milhões;
- o foquete Ariane 5 explodiu na plataforma de lançamento por causa de um erro de *software;*
- o bombardeio B-2 não conseguiu realizar um vôo por causa de um problema de *software;*.
- as balsas controladas por computador em Seattle causaram mais de uma dúzia de colisões nas docas, tendo por resultado os danos avaliados em mais de US$7 milhões. O estado de Washington recomendou o investimento de mais de US$3 milhões para retornar para manual os controles das balsas.

Outra grave conseqüência dos erros em *software* são os problemas de segurança. Segundo Kaner *et al* (2002, p. 73), os *bugs* muito extremos são falhas de segurança em potenciais, como afirma (Schneider 2000a and 2000b apud 21) em "A maioria das invasões pela internet levam vantagem nos erros de buffer-overrun *bugs*" (KANER *et al*, 2002, p. 73).

De acordo com o Boletin Segurança Digital (SEGURANÇA, 2003), o Brasil continua no topo da lista dos grupos hackers em todo o mundo. De acordo com pesquisa desenvolvida pela mi2g Intelligence Unit, empresa de consultoria de risco digital baseada em Londres, entre os dez grupos que mais invadiram sites durante o mês de outubro, sete estão baseados no Brasil. De acordo com a mi2g, só em 2003 a ação dos hackers já causaram prejuízos entre US$ 118,8 bilhões e US$ 145,1 bilhões (entre R$ 336 bilhões e R$ 413 bilhões) em todo o mundo (SEGURANÇA, 2003).

De acordo com outra pesquisa, a 9ª edição da Pesquisa Nacional de Segurança da Informação, realizada pela Módulo Security, 77% das empresas brasileiras identificaram ataques ou invasões a suas redes de computadores em 2003. O número era de 43% em 2002 (INFORMÁTICA, 2003).

A pesquisa "Computer Crime and Security Survey" de 2002, conduzida pelo Computer Security Institute juntamente com o FBI de São Francisco reportou as seguintes estatísticas (SPLAINE, 2002, p. 2):

- 90% dos respondentes (principalmente grandes corporações e agências de governo) detectaram quebras na segurança de computadores nos ultimos 12 meses;
- 74% dos respondentes citaram as suas conexões de internet como um frequente ponto de ataque, e 40% detectaram a penetração nos sistemas advindas do meio externo a empresa;
- 75% dos respondentes estimaram que empregados contrariados eram uma fonte provável de alguns destes ataques.

A lista apresentada a seguir mostra o número de incidentes relacionados com a segurança de sistemas de informação relatados pelo *CERT Coordination Center* [1](SPLAINE, 2002, p. 2):

[1] www.cert.org

- em 2002 (1° e 2° trimestre) - 43.136;
- em 2001 – 52.658;
- em 2000 – 21.756;
- em 1999 – 9.859;
- em 1998 – 3.734.

Em fevereiro de 2002 a Reuters[2] relatou que "*hackers*" forçaram a *CloudNine Communications*, uma dos mais antigos provedores de internet britânicos, a saírem do negócio. A empresa chegou a conclusões que o custo de se recuperar do ataque era tão grande para a empresa que ela resolveu entregar seus clientes a um concorrente. (SPLAINE, 2002, p. 2)

Humphrey (2004) afirma que hoje, em torno de 90% de todas as vulnerabilidades de segurança são variações de defeitos funcionais nos *softwares*. O autor afirma que da perspectiva da segurança, um defeito por 1000 linhas de código é totalmente inadequado, sendo que isto representa um defeito em 30 a 40 páginas impressas de código fonte de programa, nível de qualidade este, que em sua visão, nenhum produto produzido por um humano manualmente pode atingir (HUMPHREY, 2004).

Qualquer *software*, independente das tecnologias de teste utilizadas, tem *bugs*. Alguns são detectados e removidos em tempo de desenvolvimento. Outros, são encontrados e reparados durante os testes formais. No entanto, todos os produtores de *software* sabem que *bugs* permanecem no *software* e que alguns deles terão que ser corrigidos mais tarde (Beizer 1990 apud Li e Wu, 2004, p. 1). O teste é um pré-requisito necessário para a implementação bem sucedida do *software*, mas com as tecnologias de testes atualmente disponíveis, ele é considerado frequentemente difícil, entediante, consumidor de muito tempo, e inadequado. Um estudo divulgou que os *bugs* nos produtos finais custo à economia dos Estados Unidos US$59,2 bilhões por ano (Tassey 2002 apud Li e Wu, 2004, p. 1).

Apesar dos testes e a qualidade dos *softwares* não serem sinônimos, certamente o nível de qualidade dos testes de um *software* é um fator importante, entre outros, para definir a

[2] www.reuters.co.uk

qualidade do produto final, que depende do processo de desenvolvimento deste *software* (FILHO e RIOS, 2003 apud COSTA, 2003, p.12).

Os problemas com *software* custam anualmente US$59.5 bilhões para a economia americana, de acordo com um estudo do Instituto de Normas e Tecnologia do Departamento Americano de Comércio (*U.S. Department of Commerce's National Institute of Standards and Technology* – NIST). O estudo encontrou que os usuários de *software* são os responsáveis por quase 50% dos problemas, e os vendors e desenvolvedores contribuem com o resto. Por outro lado, o estudo também concluiu que uma melhoria nos testes poderia reduzir os *bugs* e os custos em US$22,2 bilhões. Além disto, o estudo também concluiu que mesmo com quase 80% do desenvolvimento de *software* sendo investido em localização e correção de defeitos, quase nenhum outro tipo de produto possui as mesmas quantidades de defeito que os *softwares* (NIST apud FISHMAN, 1996). A Academia Nacional de Ciências emitiu um relatório descrevendo que os legisladores devem aprovar uma legislação para fazer com que os vendedores de *software* sejam os responsáveis pela quebra de segurança. No entanto, esta medida aumentaria agressivamente o custo de seus produtos, de acordo com Marc E. Brown sócio da empresa McDermott, Will, e Emory. A Europa já começou a endereçar este problema quando um juiz holandês se convenceu que a Exact Holding vendia *softwares* com defeitos, negando sua reivindicação de que muitos problemas são comuns em versões preliminares de *softwares* (FISHMAN, 1996).

Sendo assim uma série de medidas analíticas devem ser feitas para avaliar os resultados das diferentes fases do desenvolvimento e detectar os erros o mais cedo possível. O teste constitui a mais importante técnica de análise, ao lado das revisões e inspeções. Ele é, no entanto, uma tarefa muito sofisticada e consumidora de tempo, particularmente no campo dos sistemas embarcados. (BROEKMAN; NOTENBOOM, 2003, p. x)

Durante o processo de desenvolvimento de um *software*, existem atividades que procuram garantir a qualidade do produto final. Entretanto, apesar dos métodos, técnicas e ferramentas utilizadas, falhas no produto ainda podem ocorrer. Assim, a etapa de teste, que representa uma das atividades de garantia de qualidade, é de grande importância para a identificação e eliminação de falhas. Teste é uma área da engenharia de *software* e tem como objetivo aprimorar a produtividade e fornecer evidências da confiabilidade e da qualidade do *software*

em complemento a outras atividades de garantia de qualidade ao longo do processo de desenvolvimento de *software* (INTHURN, 2001).

Através da gestão do processo de teste, obter-se-á um melhor acompanhamento e controle sobre as atividades desempenhadas no projeto, melhorando a qualidade e produtividade dos trabalhos *design*ados à equipe de teste, e aperfeiçoando os produtos desenvolvidos pelas demais equipes do projeto (COSTA, 2003, p. 27).

Os sistemas que antes trabalhavam isolados têm sido integrados para possibilitar novas aplicações. Desta forma, os processos de teste têm se tornado maiores, mais complexos e mais difíceis de serem controlados. (BROEKMAN; NOTENBOOM, 2003, p. x)

Os testes consomem de 25% a 50% dos orçamento total em muitos projetos de desenvolvimento. Um time de teste consiste de engenheiros que atuam como testadores manuais, usuários de ferramentas, e desenvolvedores de ferramentas. Tanto o orçamento como o pessoal são importantes pelo fato de que o produto em desenvolvimento deve ser testado tão completamente quanto possível. (LI; WU, 2004, p. 1)

1.3 Pergunta de pesquisa

Esta dissertação é fundamentada em uma revisão bibliográfica dentro dos temas garantia da qualidade de *software* e testes de *software*, bem como em uma pesquisa quantitativa composta por questões fechadas, com o intuito de responder à seguinte pergunta de pesquisa: "Quais as práticas e disciplinas utilizadas em testes bem como atributos de qualidade considerados nas empresas brasileiras para garantir a qualidade dos *softwares* desenvolvidos?"

1.4 Objetivos do estudo

O objetivo deste estudo é conhecer e traçar um panorama sobre quais são as práticas e disciplinas utilizadas nos processos de testes de *software* conduzidos pelas empresas, de diferentes setores de atuação e porte, que desenvolvem ou encomendam para outras

organizações sistemas de informação, bem como mapear qual é o grau de conhecimento dos principais atributos de qualidade de *software* e o grau de utilização formal de testes para avaliá-los. Ainda neste contexto, um desdobramento do objetivo principal é a avaliação, através da aplicação do modelo de avaliação proposto pelo QAI (*Quality Assurance Institute*), descrito na obra de William Perry ("*Effective Methods for Software Testing*", 2000), do grau de capacidade das empresas nos seguintes *drivers* de processos de teste de classe mundial: Satisfação dos Usuários, Suporte Gerencial aos Testes, Treinamento em Testes, Planejamento em Testes, Utilização de Processos de Testes, Ferramentas de Testes, Eficiência dos Testes.

2 REVISÃO BIBLIOGRÁFICA

2.1 Introdução

Este capítulo tem como finalidade apresentar os principais conceitos relazionados à qualidade de *software*, bem como fundamentar a pesquisa, a partir da revisão bibliográfica realizada.

2.2 Defeitos

Neste trabalho, apesar de serem apresentadas as diversas definições e diferenças, na visão de alguns autores, para os termos erro, falha, defeito e *bug*, eles serão considerados sinônimos pelas mesmas razões de Beizer (2000)[3], principalmente a título da pesquisa e sua análise,.

2.2.1 Conceitos

Um erro (*error*) é um engano, idéia falsa, ou mal entendimento por parte do desenvolvedor de *software*, considerando como desenvolvedor um engenheiro de *software*, um programador, um analista e um testador. Por exemplo, um desenvolvedor pode não entender uma notação do modelo, ou um programador pode digitar um nome de uma variável incorretamente (BURNSTEIN, 2003, p. 20).

As falhas (*faults*) ou defeitos (*defects*) são introduzidos no *software* como resultado de um erro. Trata-se de uma anomalia no *software* e pode provocar um comportamento incorreto e em desacordo com a sua especificação. Muitas vezes as falhas ou defeitos são chamados de *bugs* (BURNSTEIN, 2003, p. 20).

---

[3] Os motivos apresentados pelo autor são: (1) todos possuem um entendimento de que os termos são sinônimo; (2) as normas são inconsistentes entre si com relação à definição de cada um dos termos; (3) o uso da palavra bug é o mais conhecido dentro da indústria de software, trata-se um um jargão.

Uma falha (*failure*) é a inabilidade de um sistema ou componente do *software* de realizar suas funções requeridas dentro dos requisitos de performance especificados. Durante a execução de um componente ou sistema de *software*, um testador, desenvolvedor, ou usuário observa que ele não produz os resultados esperados. Em alguns casos um tipo particular de desvio de comportamento indica que um certo tipo de falha (*fault*) está presente (BURNSTEIN, 2003, p. 20).

Burnstein (2003, p. 39) afirma que mesmo sob as melhores condições de desenvolvimento os erros são criados, resultando em defeitos injetados durante as fases do ciclo de vida do *software*. Os defeitos provém das seguintes fontes: educação, comunicação, descuido, transcrição ou processo (BURNSTEIN, 2003, p. 39-40).

Em termos de educação, Burnstein (2003, p. 39-40) menciona a situação em que os desenvolvedores não possuem o embasamento educacional adequado para preparar algum artefato ou executar alguma atividade. No caso da comunicação, o autor cita a situação em que um engenheiro de *software* não é informado de algo por um colega. No caso de descuido, o desenvolvedor se omite em fazer algo. Em se tratando da transcrição, o desenvolvedor sabe o que precisa ser feito, no entanto comete um erro na execução, como no caso, por exemplo, de escrever um comando com a sintaxe errada. Os processos guiam as ações dos engenheiros de *software*, portanto, eles podem orientar incorretamente as ações dos recursos, como, por exemplo, no caso do processo de desenvolvimento não deixar tempo suficiente para a especificação detalhada ser desenvolvida e revisada, podendo resultar em defeitos de especificação (BURNSTEIN, 2003, p. 39-40).

Beizer (1990, p. 19) afirma que os *bugs* são sempre mais incidentes do que se espera, e que a importância deles depende (BEIZER, 1990, p. 28) da freqüência, custo de reparo, custo de instalação, e conseqüências. A freqüência está relacionada com a quantidade de vezes que um dado tipo de *bug* ocorre. O custo de correção é a soma de dois fatores, o custo de descoberta e o custo de correção. Este custo, segundo Beizer (1990, p. 28) cresce dramaticamente ao longo do ciclo de desenvolvimento. Os custos de correção também dependem do tamanho do sistema, já que quanto maior for este, maior o custo de correção de um mesmo *bug*. O custo de instalação depende do número de instalações. O autor cita a possibilidade do custo de correção de um simples *bug* juntamente com a distribuição desta correção exceder o custo de todo o desenvolvimento do sistema. As conseqüências do *bug*, segundo o autor, podem ser

mensuradas pelo tamanho médio das concessões feitas pelos júris às vítimas de seu erro. (BEIZER, 1990, p. 28)

O autor define uma métrica para mensurar a importância de um *bug*: (BEIZER, 1990, p. 28)

Importância ($) = freqüência x (custo de correção + custo de instalação + custo da conseqüência)

A freqüência, segundo Beizer (1990, p. 29), geralmente não depende da aplicação ou do ambiente, no entanto os custos de correção, de instalação e das conseqüências sim. Ela apresenta uma escala para classificação das conseqüências de um *bug*. De acordo com Beizer (1990, p. 29-32), as conseqüências podem ser de suaves (*mild*) à catastróficas (*catastrophic*). Sua escala de conseqüências possui 10 níveis: suave, moderado, irritante (*annoying*), distúrbio (*disturbing*), sério, muito sério, extremo (*extreme*), intolerável, catastrófico e infeccioso (*infectious*).

2.2.2 Taxonomia dos defeitos

Alguns autores adotam o termo "modelo de falha" (*fault model*) para a classificação dos defeitos de forma ordenada, agrupada e com relacionamentos hierárquicos (*Testing Object-Oriented Systems*, Robert Binder apud 15) (BURNSTEIN, 2003, p. 42). Outros autores adotam o termo "Taxonomia de defeitos" (COPELAND, 2004; BEIZER, 1990, p. XX). Burnstein (2003, p. 39) também utiliza a expressão "*defect model*", além de "*fault model*".

Copeland (2004) define a taxonomia como uma classificação das coisas em grupos ordenados, que indicam relacionamentos hierárquicos e naturais.

Burnstein (2003, p. 42) diz que um modelo de falha (*fault model*) pode ser descrito como uma relação entre o erro cometido, como por exemplo, um requisito não presente ou um erro de digitação, e o defeito no *software*.

Segundo Copeland (2004), a taxonomia de *bugs* não somente facilita o armazenamento organizado destas informações, como também ajuda a:

- orientar o teste através da geração de idéias para casos de teste;
- auditar os planos de testes para determinar a cobertura que os casos de teste estão provendo;
- entender os defeitos, bem como seus tipos e gravidades (severidades);
- entender o processo utilizado para produzir aqueles defeitos;
- melhorar o processo de desenvolvimento;
- melhorar o processo de teste;
- treinar novos testadores de acordo com áreas importantes que necessitam de testes;
- e explicar à administração a complexidade do teste de *software*.

Em princípio, uma taxonomia de defeito atua como uma lista de verificação (*checklist*), lembrando ao testador que nenhum tipo de defeito foi esquecido (COPELAND, 2004).

Outra aplicação ocorre numa situação em que existam muitas coisas para serem testadas e o tempo é insuficiente. Neste caso, os dados de uma taxonomia possibilitam a realização de decisões de modelagem de teste baseadas em risco, ao invés de randômicas (COPELAND, 2004).

Beizer (1990, p. 33) menciona que não existe uma forma considerada universalmente correta de categorizar os *bugs*, pois eles são difíceis de serem classificados, já que um dado *bug* pode ser colocado em diversas categorias dependendo de seu histórico e do estado da mente do programador.

Beizer (1990, p. 33) ressalta ainda que a taxonomia dos defeitos, assim como o teste, é potencialmente infinita e que, mais importante do que adotar a taxonomia "correta", é a adoção de alguma taxonomia bem como a sua utilização com uma estrutura estatística para apoiar a estratégia de teste.

Robert Binder, citado por Copeland (2004), inclusive separa as abordagens de teste de *software* de acordo com a utilização ou não de um "modelo de falha". Segundo o autor, existem duas abordagens, onde uma delas utiliza um "modelo de falha não específico", ou seja, não utiliza um modelo de falha ou taxonomia de defeitos, de forma que os requisitos e demais especificações é que orientam a criação de todos os casos de teste, e a outra delas

utiliza um "modelo de falha específico", onde a taxonomia do defeito orienta a criação dos casos de teste (COPELAND, 2004).

Copeland (2004) apresenta uma série de diferentes taxonomias de defeitos: de Beizer, de Kaner, Falk, e Nguyen, de Binder, de Whittaker e de Vijayaraghavan. O autor chama a atenção para o fato de nem todas as taxonomias apresentadas serem completas, pois algumas delas podem ser expandidas. Além disto, ele ressalta que cada uma é subjetivamente baseada na experiência de seus criadores.

Uma das primeiras taxonomias de defeitos, segundo Copeland (2004), foi criada por Boris Beizer em *Software Testing Techniques*. Ela define uma classificação dos defeitos de *software* de quatro níveis. Os dois níveis mais altos são mostrados na Tabela 1 (COPELAND, 2004).

As principais categorias são: requisitos (*requirements*), características e funcionalidades (*features* e *functionality*), estrutura (*structure*), dado (*data*), implementação e codificação (*implementation* e *coding*), integração (*integration*), arquitetura de sistema e de *software* (*system* e *software architecture*) e teste (*testing*) (BEIZER, 1990, p. 58).

Tabela 1 - Parte da taxonomia de *bugs* de Beizer[4]

1xxx	**Requirements**
11xx	Requirements incorrect
12xx	Requirements logic
13xx	Requirements, completeness
14xx	Verifiability
15xx	Presentation, documentation
16xx	Requirements changes
2xxx	Features And Functionality
21xx	Feature/function correctness
22xx	Feature completeness
23xx	Functional case completeness
24xx	Domain *bugs*
25xx	User messages and diagnostics
26xx	Exception conditions mishandled
3xxx	Structural *Bugs*
31xx	Control flow and sequencing
32xx	Processing
4xxx	Data

[4] A tabela foi mantida em seu idioma original, pelo fato da tradução de alguns termos, eventualmente, gerar algum tipo de prejuízo ao entendimento. Como os ambientes de desenvolvimento e sistemas operacionais apresentam estes termos em suas mensagens de erro, uma tradução poderia acarretar em uma perda de precisão.

41xx	Data definition and structure
42xx	Data access and handling
5xxx	Implementation And Coding
51xx	Coding and typographical
52xx	Style and standards violations
53xx	Documentation
6xxx	Integration
61xx	Internal interfaces
62XX	External interfaces, timing, throughput
7XXX	System And *Software* Architecture
71XX	O/S call and use
72XX	*Software* architecture
73XX	Recovery and accountability
74XX	Performance
75XX	Incorrect diagnostics, exceptions
76XX	Partitions, overlays
77XX	Sysgen, environment
8XXX	Test Definition And Execution
81XX	Test *design bugs*
82XX	Test execution *bugs*
83XX	Test documentation
84XX	Test case completeness

FONTE: COPELAND, 2004

A Taxonomia de Kaner, Falk e Nguyen é apresentada no livro *Testing Computer Software* e contém uma lista detalhada com mais de 400 tipos de defeitos. Somente uma porção desta taxonomia é apresentada na Tabela 2. (COPELAND, 2004)(colocar a referência direta de Kaner *et al*)

Tabela 2 - Parte da taxonomia de defeitos de Kaner *et al* [5]

User Interface Errors	Functionality
	Communication
	Command structure
	Missing commands
	Performance
	Output
Error Handling	Error prevention
	Error detection
	Error recovery
Boundary-Related Errors	Numeric boundaries
	Boundaries in space, time
	Boundaries in loops
Calculation Errors	Outdated constants
	Calculation errors
	Wrong operation order

[5] Tabela mantida em seu idioma original pelos mesmos motivos apresentados anteriormente.

Initial And Later States	Overflow and underflow
	Failure to set a data item to 0
	Failure to initialize a loop control variable
	Failure to clear a string
	Failure to reinitialize
Control Flow Errors	Program runs amok
	Program stops
	Loops
	IF, THEN, ELSE or maybe not
Errors In Handling Or Interpreting Data	Data type errors
	Parameter list variables out of order or missing
	Outdated copies of data
	Wrong value from a table
	Wrong mask in bit field
Race Conditions	Assuming one event always finishes before another
	Assuming that input will not occur in a specific interval
	Task starts before its prerequisites are met
Load Conditions	Required resource not available
	Doesn't return unused memory
Hardware	Device unavailable
	Unexpected end of file
Source And Version Control	Old *bugs* mysteriously reappear
	Source doesn't match binary
Documentation	None
Testing Errors	Failure to notice a problem
	Failure to execute a planned test
	Failure to use the most promising test cases
	Failure to file a defect report

FONTE: COPELAND, 2004

Robert Binder (apud 15) observa que muitos defeitos na orientação a objetos são problemas utilizando encapsulamento, heranças, polimorfismo, sequenciamento de mensagem, e transições de estado. Isto é esperado por duas razões, segundo o autor: primeiro pelo fato de serem conceitos chave na Orientação a Objeto e segundo pelo fato destes conceitos serem muito diferentes da programação procedural. Uma pequena porção da Taxonomia de Orientação a objetos de Binder é mostrada na

Tabela 3 e na Tabela 4. (COPELAND, 2004) (ir direto à fonte)

Tabela 3 – Uma parte da taxonomia de Binder de erro de escopo do método (Binder's Method Scope Fault Taxonomy) [6]

Method Scope		Fault
Requirements		Requirement omission
Design	Abstraction	Low Cohesion
	Refinement	Feature override missing
		Feature delete missing
	Encapsulation	Naked access
		Overuse of friend
	Responsibilities	Incorrect algorithm
		Invariant violation
	Exceptions	Exception not caught

FONTE: COPELAND, 2004

Tabela 4 - Uma parte da taxonomia de Binder de erro de escopo da classe (Binder's Class Scope Fault Taxonomy) [7]

Class Scope		Fault
Design	Abstraction	Association missing or incorrect
		Inheritance loops
	Refinement	Wrong feature inherited
		Incorrect multiple inheritance
	Encapsulation	Public interface not via class methods
		Implicit class-to-class communication
	Modularity	Object not used
		Excessively large number of methods
	Implementation	Incorrect constructor

FONTE: LI e WU, 2004

Em seu livro *How to Break Software*, James Whittaker não somente identifica as áreas onde as falhas tendem a ocorrer, como define testes específicos para localizar estas falhas. Uma parte de sua da taxonomia é mostrada na Tabela 5 (COPELAND, 2004). (usar a referência do autor diretamente)

Tabela 5 – Parte da taxomonia de erros de Whittaker [8]

Fault Type	Attack
Inputs and outputs	Force all error messages to occur
	Force the establishing of default values
	Overflow input buffers
Data and computation	Force the data structure to store too few or too many values
	Force computation results to be too large or too small
File system interface	Fill the file system to its capacity

[6] Mantida no idioma original pelas mesmas razões apresentadas anteriormente.
[7] Mantida no idioma original pelos motivos apresentados anteriormente.
[8] Idem nota de rodapé anterior.

	Damage the media
Software interfaces	Cause all error handling code to execute
	Cause all exceptions to fire

FONTE: COPELAND, 2004

Burnstein (2003, p. 39-51) apresenta uma outra taxonomia de defeitos em sua obra *Practical Software Testing – A process-oriented approach* composta por quatro categorias principais de defeito classificadas de acordo com cada uma das quatro principais etapas de desenvolvimento de um sistema: Defeitos de Requisitos e Especificação, Defeitos de Modelagem (*Design*), Defeitos de Codificação e Defeitos de Testes. A Tabela 6 lista a taxonomia de defeitos de Burnstein.

Tabela 6 - Taxonomia de defeitos de Burnstein [9]

Requirements and Specification Defects
Functional Description Defects
Feature Defects
Feature Interaction Defects
Interface Description Defects
***Design* Defects**
Algorithmic and Processing Defects
Control, Logic, and Sequence Defects
Data Defects
Module Interface Description Defects
Functional Description Defects
External Interface Description Defects
Coding Defects
Algorithmic and Processing Defects
Control, Logic and Sequence Defects
Typographical Defects
Initialization Defects
Data-Flow Defects
Data Defects
Module Interface Defects
Code Documentation Defects
External Hardware, *Software* Interfaces Defects

[9] Mantida no idioma original pelas razões já apresentadas.

Testing Defects

Test Harness Defects

Test Case *Design* and Test Procedure Defects

FONTE: adaptado de Burnstein (2003, p. 39-51)

As taxonomias de Beizer, Kaner e Whittaker catalogam defeitos que podem ocorrer em qualquer sistema, enquanto Binder foca nos defeitos comuns em sistemas orientados a objeto. Já Giri Vijayaraghavan escolheu um foco mais estreito – *websites* de comércio eletrônico (*e-commerce*). Vijayaraghavan investigou as diversas formas na quais tais aplicações podem falhar. Sua taxonomia lista mais de sessenta categorias de defeitos em alto-nível, dentre elas: performance, confiabilidade, atualizações de *software*, usabilidade da interface do usuário, manutenabilidade, conformidade, estabilidade, operabilidade, tolerância à falhas, acurácia, internacionalização, recuberabilidade, planejamento de capacidade, falhas de *software* de terceiros, vazamentos de memória, problemas de navegador (*browser*), segurança do sistema, e privacidade do cliente (COPELAND, 2004).

Por fim, Copeland (2004) observa que cada uma destas taxonomias é uma lista de possíveis defeitos sem qualquer direcionamento em relação a probabilidade de ocorrência deles nos sistemas e sem qualquer sugestão da perda que pode ser ocasionada na organização se qualquer um destes defeitos ocorrer. As taxonomias são pontos de início úteis para os testes, mas certamente não são a resposta completa à questão de onde iniciar o teste.

2.2.3 Estatísticas

Beizer (1990) menciona que as estatísticas mostram que os defeitos ocorrem aproximadamente na seguinte distribuição:

- Requisitos e Características – 50%
- Funcionalidade da forma como foi implementada – 10%
- *Bugs* estruturais – 10%
- Implementação – 9%
- Dados – 4%
- Integração – 7%

- Sistema Operacional e Tempo Real – 6%
- Definição de Teste ou *Bug* de execução – 4%

Beizer (1990, p. 56-58) menciona que a freqüência dos *bugs* obtida a partir de diversas fontes mostra que ocorrem aproximadamente 2.4 *bugs* por 1000 linhas de código fonte. Como alguns exemplos não incluem linhas não executáveis, o valor real é provavelmente mais baixo.

O autor explica que o fato desta taxa de 0,24% ser diferente da taxa 1%-3%, geralmente citada, deve-se ao fato desta última levar em consideração todos os *bugs* encontrados, desde o teste unitário até a utilização em campo, e, tipicamente incluir todos os *bugs* descobertos pelos programadores durante seus próprios testes e inspeções, enquanto que a primeira leva em consideração apenas os testes independentes, testes de integração e de sistema, que ocorrem após o teste de componentes realizado pelos programadores (BEIZER, 1990, p. 56-58).

Beizer (1990, p. 58) menciona que a importância da Tabela 7, que apresenta estatísticas de *bugs* distribuídos utilizando a sua taxonomia, não está na freqüência absoluta dos *bugs* (ou seja, número de *bugs* por milhares de linhas de código) e sim na freqüência relativa destes *bugs* por categoria.

Tabela 7- Exemplo de estatísticas de *BUG* [10]

SIZE OF SAMPLE—6,877,000 STATEMENTS (COMMENTS INCLUDED)
TOTAL REPORTED *BUGS*—16,209—*BUGS* PER 1000 STATEMENTS—2.36

1xxx REQUIREMENTS	1317	8.1%
11xx Requirements Incorrect	649	4.0%
12xx Requirements Logic	153	0.9%
13xx Requirements, Completeness	224	1.4%
15xx Presentation, Documentation	13	0.1%
16xx Requirements Changes	278	1.7%
2xxx FEATURES AND FUNCTIONALITY	2624	16.2%
21xx Feature/Function Correctness	456	2.8%
22xx Feature Completeness	231	1.4%

[10] Mantida no idioma original pelas razões já apresentadas.

23xx Functional Case Completeness	193	1.2%
24xx Domain *Bugs*	778	4.8%
25xx User Messages and Diagnostics	857	5.3%
26xx Exception Condition Mishandled	79	0.5%
29xx Other Functional *Bugs*	30	0.2%
3xxx STRUCTURAL *BUGS*	4082	25.2%
31xx Control Flow and Sequencing	2078	12.8%
32xx Processing	2004	12.4%
4xxx DATA	3638	22.4%
41xx Data Definition and Structure	1805	11.1%
42xx Data Access and Handling	1831	11.3%
49xx Other Data Problems	2	0.0%
5xxx IMPLEMENTATION AND CODING	1601	9.9%
51xx Coding and Typographical	322	2.0%
52xx Style and Standards Violations	318	2.0%
53xx Documentation	960	5.9%
59xx Other Implementation	1	0.0%
6xxx INTEGRATION	1455	9.0%
61xx Internal Interfaces	859	5.3%
62xx External Interfaces, Timing, Throughput	518	3.2%
69xx Other Integration	78	0.5%
7xxx SYSTEM, *SOFTWARE* ARCHITECTURE	282	1.7%
71xx O/S Call and Use	47	0.3%
72xx *Software* Architecture	139	0.9%
73xx Recovery and Accountability	4	0.0%
74xx Performance	64	0.4%
75xx Incorrect Diagnostics, Exceptions	16	0.1%
76xx Partitions, Overlays	3	0.0%
77xx Sysgen, Environment	9	0.1%
8xxx TEST DEFINITION AND EXECUTION	447	2.8%
81xx Test *Design Bugs*	11	0.1%
82xx Test Execution *Bugs*	355	2.2%
83xx Test Documentation	11	0.1%
84xx Test Case Completeness	64	0.4%
89xx Other Testing *Bugs*	6	0.0%
9xxx OTHER, UNSPECIFIED	763	4.7%

FONTE: BEIZER 1990

Beizer (1990, p. 11) descreve ainda que as técnicas de teste e programação melhoraram, de forma que os *bugs* se deslocaram em direção às etapas iniciais, como os requisitos e sua especificação. Estes *bugs* variam entre 8 a 30% do total de erros encontrados, e, pelo fato deles serem os "primeiros a entrar" e os "últimos a sair", são os com maiores custos.

2.3 Qualidade

2.3.1 Definição da Qualidade

Na literatura especializada, são encontradas inúmeras diferentes definições de qualidade. (KAN, 2003, p. 1-3; LEWIS, 2000, p. 3; INTHURN, 2001, p. 5). Além disto, segundo Kan (2003, p. 1), o termo "qualidade" é ambíguo, e, por isto é geralmente confundido. Kan (2003, p. 1) afirma que a confusão pode ser atribuída a diversas razões. A primeira delas, segundo o autor, é que qualidade não é uma idéia simples, e sim um conceito multidimensional. A segunda é o fato de em qualquer conceito existem níveis de abstração, ou seja, quando alguém fala sobre qualidade, uma parte pode estar se referindo ao seu sentido mais amplo, enquanto que a outra pode estar se referindo a um significado específico. A terceira razão é que o termo qualidade é uma parte de nossa linguagem cotidiana, de forma que o uso popular e profissional deste pode variar muito.

Esta variação fica evidente quando se compara algumas diferentes definições. Para o fundador de uma loja de departamentos, Herrmann Titz, qualidade significa o cliente voltar à loja e não o produto (DE YOUNG; RÄTZMANN, 2003, p. 25). Já a norma DIN 55350 da industria alemã, citada por DeYoung e Rätzmann (2003, p.25), define qualidade como o agregado de todas as características e propriedades de um produto ou atividade que se relacionam com sua capacidade de atender requisitos especificados. No dicionário Webster, citado por Lewis (2000), a qualidade é definida como "a característica essencial de algo, uma característica inerente ou que distingua, grau de excelência". Outra definição é a de Weinberg (1992), citado por Kaner *et al* (2002), onde a qualidade é "... valor para alguma pessoa".

Lewis (2000, p. 3) menciona que a literatura computacional traz dois conceitos geralmente aceitos de qualidade. No primeiro, qualidade significa "atender requisitos". Dentro deste conceito, para se ter um produto de qualidade, os requisitos devem ser mensuráveis, para que o atendimento seja mensurável. Dentro desta definição, qualidade é, tal como Lewis (2000, p. 3) descreve, um estado binário, ou seja, o produto é de qualidade ou não. Os requisitos podem ser bem completos ou simples, no entanto, em sendo mensuráveis, pode-se determinar quando a qualidade foi ou não alcançada. O autor diz que esta é a visão de qualidade do produtor –

em atender os requisitos ou especificações do produto – onde atender as especificações se torna, por si mesmo, o fim (LEWIS, 2000, p. 3-4).

A outra definição de qualidade é a do cliente, a preferida de Lewis (2000, p.3). Nesta, a qualidade de um produto ou serviço está vinculada ao atendimento das necessidades do cliente, ou como o autor menciona, "adequação ao uso" (LEWIS, 2000, p. 3).

Kan (2003, p. 1) classifica as diferentes visões da qualidade em duas categorias: popular e profissional. Na primeira, a qualidade é algo intangível, já que a mesma pode ser discutiva, percebida, julgada, mas não pode ser medida ou ponderada. Dentro desta visão está o uso vago da palavra qualidade em expressões como "boa qualidade", "péssima qualidade" ou mesmo "qualidade de vida". Esta visão reflete o fato, segundo o autor, das pessoas perceberem e interpretarem qualidade de maneiras diferentes. O desdobramento disto é o fato da qualidade não poder ser controlada ou gerenciada, nem ao menos quantificada. Ainda dentro do conceito popular, uma outra visão descrita pelo autor é aquela que conota luxo, classe e bom gosto. Neste contexto, produtos caros, elaborados e mais complexos são tidos como detentores de um elevado nível de qualidade. Dentro desta visão, a qualidade se restringe a um conjunto limitado de produtos caros, com funcionalidades sofisticadas e itens com um toque de classe. Produtos baratos e simples dificilmente podem ser classificados como detentores de qualidade (KAN, 2003, p. 1-2).

Já no âmbito profissional, Kan (2003, p. 2-4) menciona que a confusão e imprecisão da visão popular não se aplica, pois não ajuda na melhoria da qualidade. No contexto profissional, qualidade, em sua visão, deve ser descrita em termos de uma definição que funcione para tal fim, e para tal, ele cita a definição de Crosby (1979), citado por Kan (2003, p. 2-4) – "conformidade com os requisitos" – e a de Juran e Gryna (1970), citado por Kan (2003, p. 2-4) – "adequação ao uso", as quais, segundo o autor, estão relacionadas e são consistentes, e, por este motivo, têm sido adotadas e utilizadas por muitos profissionais.

Segundo Kan (2003, p.1) há um grande contraste entre a visão popular e a profissional, já que nesta ultima a qualidade deve ser definida operacionalmente, mensurada, monitorada, gerenciada e melhorada.

Na indústria do *software*, a definição de qualidade relacionada com a conformidade aos requisitos do cliente é muito relevante, já que os erros de requisitos constituem uma das categorias de maior problema. De acordo com Jones (1992), citado por Kan (2003, p. 5-6), 15% ou mais de todos os defeitos do *software* são erros de requisitos (KAN, 2003, pg. 5-6). Craig e Jaskiel (2002) apresentam as definições clássicas de Philip Crosby e de Dr. Joseph M. Juran. Para Crosby, de acordo com a descrição dos autores, a qualidade é a conformidade com os requisitos, sendo que a falta de conformidade significa falta de qualidade. De acordo com os autores, para Juran a qualidade se traduz na presença daquilo que satisfaz os clientes e usuários e a ausência daquilo que insatisfaz.

2.3.2 Qualidade de *Software*

A qualidade, no âmbito específico do produto *software*, encontra algumas definições na literatura especializada. Uma delas, a de Hutcheson (2003, p. 43), é que a qualidade do *software* é "uma combinação de confiabilidade, rapidez na oferta ao mercado, preço/custo, e a riqueza de características". Outras duas definições concisas para qualidade de *software*, citada em Burnstein (2003, p. 23), são encontradas no IEEE *Standard Glossary of Software Engineering Terminology*:

- Qualidade está relacionada ao grau no qual um sistema, componente do sistema, ou processo que atende requisitos especificados.
- Qualidade está relacionada ao grau no qual um sistema, componente do sistema, ou processo atende as necessidades ou expectativas dos clientes ou expectativas.

Ainda de acordo com o IEEE *Computer Society glossary*, tal como citado em De Young e Rätzmann (2003, p. 27), a qualidade do *software* é:

a) A totalidade de funcionalidades e características de um *software* que estão relacionadas com sua habilidade de satisfazer dadas necessidades; por exemplo, conformidade com especificações;
b) O grau no qual o *software* possui uma combinação desejada de atributos;

c) O grau com o qual um cliente ou usuário percebe que o *software* atende suas expectativas;

d) As características do *software* que determinam o grau no qual o *software* em uso irá atender as expectativas de seus clientes;

e) Atributos de um *software* que afetam seu valor percebido, como por exemplo, corretude, confiabilidade, manutenabilidade, e portabilidade;

f) Qualidade de *software* inclui adequação ao propósito, custo razoável, confiabilidade, facilidade de uso para aquele que o utiliza, características de projeto de manutenção e atualização, e o *software* se posicionar bem quando comparado com produtos confiáveis.

Bartié (2002, p. 16) diz que a qualidade de software é um processo sistemático que facaliza todas as etapas e artefatos produzidos com o objetivo de garantir a conformidade de processos e produtos, previnindo e eliminando defeitos.

Kaner *et al* (1999, p. 59) afirma que a qualidade de um *software* depende das características que fazem com que um cliente deseje utilizá-lo, bem como das falhas que fazem com que o cliente deseje ter comprado outro. Esta definição de qualidade de *software*, trazida por Kaner *et al* (1999, p. 59), leva em conta as falhas ou "defeitos" que reduzem o valor percebido do *software* por parte do cliente.

Kan (2003, p.4) também menciona esta relação entre a qualidade do *software* e os *bugs* existentes no produto, já que, como o autor descreve, trata-se de um significado básico de adequação aos requisitos, pois se o *software* contem muitos defeitos funcionais, o requisito básico de prover a funcionalidade desejada não está sendo atendido. Ele traduz ainda a sua definição de duas formas quantitativas: taxa de defeito (p.e. número de defeitos por milhões de linhas de código fonte, ou outra unidade) e confiabilidade (p.e. número de falhas por n horas de operação, MTBF – *mean time between failure*, etc).

Kan (2003, p. 6) introduz o conceito da qualidade do processo utilizado no desenvolvimento de *software* confrontando-o com a visão da qualidade do produto final. O autor menciona que desde os requisitos do cliente até a entrega do *software*, o processo de desenvolvimento é complexo e envolve uma série de estágios, cada um com realimentações. Em cada estágio, um produto intermediário é produzido para um usuário intermediário – o próximo estágio.

Cada estágio também recebe um produto intermediário de seu estágio predecessor. Cada produto intermediário possui certos atributos de qualidade que afetam a qualidade do produto final. A Figura 1 mostra uma representação simplificada de um processo comum de desenvolvimento de *software*, o processo em cascata ou *waterfall* com estas relações entre os estágios.

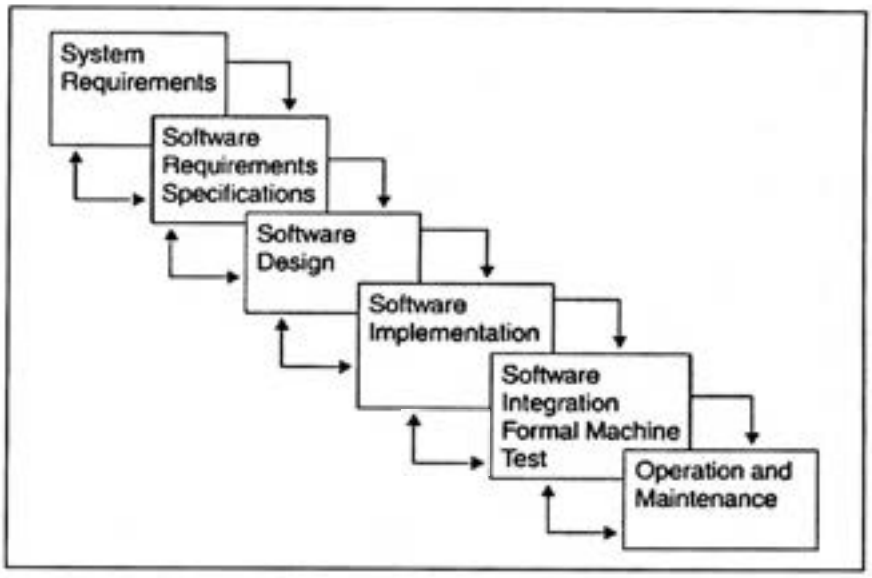

Figura 1 – Representação simplificada do processo de desenvolvimento em cascata (*Waterfall*)
FONTE: KAN, 2003, p. 6

Se o conceito de cliente na definição de qualidade for estendido para incluir os clientes internos e externos, a definição também se aplica para qualidade de processo, pois se cada estágio do processo de desenvolvimento atender aos requisitos de seu usuário intermediário (o próximo estágio), o produto final atenderá aos requisitos especificados na visão de Kan (2003). O autor reconhece no entanto que este raciocínio é uma grande simplificação da realidade, já que existem uma série de fatores que afetam a capacidade do estágio em atender seus requisitos. Ele cita ainda que para aumentar a qualidade durante o desenvolvimento, são necessários modelos do processo de desenvolvimento, e dentro do processo é necessário selecionar e implantar métodos e abordagens específicas, bem como empregar ferramentas e tecnologias apropriadas. São necessárias medidas de características e parâmetros da qualidade do processo de desenvolvimento e de seus estágios, bem como métricas e modelos, para

assegurar que o processo está sob controle e caminhando em direção aos objetivos de qualidade do produto (KAN, 2003).

Uma visão mais inovadora e recente da qualidade de *software* é descrita pelo livro *Process and Product Orientation in Software Development and their Effect on Software Quality Management*, (Wiecz e Meyerh, 2001 apud DE YOUNG; RÄTZMANN, 2003, p. 28). Esta visão toma lugar nas situações em que um processo adaptativo de desenvolvimento de *software* é utilizado no lugar de um processo transformativo. Enquanto este ultimo aplica-se em situações bem delimitadas, onde o escopo do problema é bem entendido, os requisitos bem mapeados e relativamente estáveis, bem como a flexibilidade fica em segundo plano, o processo adaptativo é utilizado nas situações em que o *software* é tão inovador que os requisitos do sistema e os problemas dos usuários estão ainda para serem pesquisados ou ainda, quando a área de aplicação deste *software* é tão caótica que o processo alvo de sua automação não pode ser identificado, já que os processos coincidem em um nível elevado, mas possuem variações no nível dos detalhes. Na visão do autor, esta ultima é, provavelmente, a causa mais comum, pois está presente nos desenvolvimentos de *software* de "prateleira", quando se deve atender diversos clientes e, ao mesmo tempo, ser compatível com seus fluxos de trabalho (DE YOUNG; RÄTZMANN, 2003, p. 28).

Nestas situações, os autores afirmam que a qualidade tem pouca relação com o "comportamento correto", já que ninguém sabe o que deve ser considerado um "comportamento correto", bem como falha o conceito de qualidade atrelado ao atendimento de requisitos, pois os requisitos funcionais detalhados não foram ainda especificados (ou pelo fato dos requisitos funcionais dos diversos clientes diferirem no nível de detalhe). Uma elevada qualidade do *software*, neste contexto, está mais relacionada à sua utilidade, à facilidade de uso, ao funcionamento de seu modelo abstrato, à sua flexibilidade e à sua capacidade de adaptar-se a requisitos desconhecidos. É importante ressaltar que estes pontos, segundo os autores, tornam-se aparentes somente depois que o *software* é lançado (DE YOUNG; RÄTZMANN, 2003, p. 28).

2.3.3 Custos da Qualidade

Os custos da qualidade variam significativamente de acordo com o produto ou serviço a ser desenvolvido, tal como descreve Beizer (1990, p. 2) ao afirmar que os custos de teste e garantia da qualidade para itens manufaturados pode ser tão baixos quanto 2% em produtos para consumo ou tão altos quanto 80% em produtos como ônibus espaciais, reatores nucleares, aviões, onde falhas ameaçam vidas. Em se tratando de *software*, seus custos são dominados pelo desenvolvimento. Sua manutenção, diferentemente do hardware, não é propriamente uma "manutenção", e sim uma extensão do desenvolvimento na qual melhorias são projetadas e implementadas, bem como deficiências são corrigidas. A maior parte do custo do *software* é o custo dos erros: o custo em detectá-los, corrigí-los, planejar testes que os revelem, e o custo de executar estes testes. (BEIZER, 1990, p. 2-3)

O gerenciamento da qualidade, segundo Lewis (2000, p. 6), reduz os custos de produção, pois, quanto antes um defeito é encontrado e corrigido, menor é o custo. Lewis (2000, p. 6) afirma ainda que, embora os investimentos iniciais possam ser substanciais, o resultado de longo prazo será a obtenção de produtos de qualidade mais elevada e custos de manutenção reduzidos (LEWIS, 2000, p. 6).

De acordo com Pressman e Humphrey, citados em Burnstein (2003, p. 395), os custos da qualidade podem ser decompostos em três grandes áreas: prevenção, avaliação, e falha. Os custos de encontrar e corrigir um defeito aumentam rapidamente a medida que se caminha por estas "fases" respectivamente. De acordo com eles, os custos de prevenção estão associados com atividades que identificam a causa dos defeitos e às ações que são tomadas para evitá-los. Como parte dos custos de preensão, Pressman: planejamento de qualidade, equipamentos de laboratório e de teste, treinamento, revisões técnicas formais. Segundo Burnstein (2003, p. 396), Pressman também incluiria neste grupo as atividades relacionadas com análise causal de defeito e prevenção.

Lewis (2000, p. 5) afirma que o maior retorno ocorre com a prevenção. Segundo o autor, aumentar a ênfase na prevensão reduz o número de defeitos não detectados que vai para o cliente, aumenta a qualidade do produto, e reduz o custo de produção e manutenção.

Os custos de avaliação envolvem aqueles associados à avaliação do *software* para determinar seu nível de qualidade. De acordo com Pressman (apud Burnstein, 2003), estes são compostos

por: testes, equipamento de calibração e manutenção, inspeções intra-processuais e inter-processuais. As revisões e verificações entrariam nesta categoria.

A verificação consiste em provar que um produto atende os requisitos especificados durante atividades prévias durante o ciclo de vida de desenvolvimento, enquanto a validação checa se o sistema atende os requisitos do cliente ao final do ciclo. Quando a verificação está incorporada no teste, este ocorre durante o ciclo de vida de desenvolvimento. (LEWIS, 2000, p. 5-6).

Um crítica da atividade de verificação é que ela aumenta os custos de desenvolvimento de *software* consideravelmente. No entanto, Lewis (2000, p. 6) comenta que quando os custos de todo o ciclo de vida do *software* são considerados, ou seja, desde sua criação até o seu total abandono, a verificação na verdade reduz o custo do *software*. Com um programa efetivo de verificação, o autor afirma que ocorre uma redução de 4 para 1 nos defeitos em um sistema instalado. Pelo fato das correções de erros custarem de 20 a 100 vezes mais durante a operação e manutenção do que na modelagem, a economia geral compensa de longe a despesa extra inicial em sua visão (LEWIS, 2000, p. 6).

Os custos de falha são aqueles que não existiriam se o *software* não tivesse defeitos. Pressman (apud BURNSTEIN, 2003, p. 396) dividiu os custos de falha em duas categorias: interno e externo. Os custos internos de falha ocorrem quando um defeito é detectado antes da entrega do *software* ao cliente. Estes custos incluem: diagnósticos ou análises do modo de falha, reparo, testes de regressão e retrabalho. Os custos externos de falha ocorrem quando o defeito é encontrado em um *software* após este ter sido entregue ao cliente. Estes incluem: resução do problema, devolução e substituição do *software*, suporte, utilização da garantia, etc (BURNSTEIN, 2003, p. 396).

Os custos de ausência de qualidade incluem teste custos de falha. Burnstein (2003, p. 396) também incluiria aos custos de ausência de qualidade aqueles associados à responsabilidade, que pode ocorrer se o *software* causar danos, perda de vida ou material. Outros custos associados com a ausência de qualidade são a insatisfação do cliente e a perda de participação de mercado (BURNSTEIN, 2003, p. 396).

Segundo Lewis (2000, p. 6), quanto mais tarde as falhas são descobertas, mais devastadoras elas podem ser pelo fato de danificarem a reputação da organização ou resultar em perdas nas vendas futuras.

2.3.4 Garantia da Qualidade

As relações existentes entre garantia da qualidade, controle da qualidade, auditoria e teste de *software* são muitas vezes confusas na literatura, conforme observa Lewis (2000, p. 8).

Hutcheson (2003, p. 25) menciona que a definição tradicional da norma britânica BS 4778 diz que a garantia da qualidade é o conjunto de todas as ações planejadas e sistemáticas necessárias para prover confiança suficiente que um produto ou serviço irá satisfazer dados requerimentos para qualidade.

Lewis (2000, p. 6) define a Garantia da Qualidade, dentro da mesma filosofia, como o conjunto de atividades de suporte necessárias para prover a confiança adequada com que os processos são estabelecidos e melhorados continuamente, de forma a produzir produtos que atendam as especificações e que sejam adequados ao uso. Para ele, a garantia da qualidade é a função responsável pelo gerenciamento da qualidade (LEWIS, 2000, p. 6).

No entanto, os conceitos tradicionais da garantia da qualidade não são inteiramente suficientes e adequados para a garantia da qualidade de *softwares*. Hutcheson (2003, p. 43) justifica isto pelo fato dos principios tradicionais da garantia da qualidade serem inflexíveis, bem como as ferramentas tradicionais utilizadas para a garantia da qualidade não serem aptas a acompanhar o desenvolvimento de *software*.

No campo do desenvolvimento de *software* Schulmeyer (apud CRSIP, 2000, p. 1) define a garantia da qualidade de *software* como "... o conjunto de atividades sistemáticas que provem a evidência da habilidade do processo de *software* de produzir um *software* que seja adequado para o uso."[11]

[11] G. Gordon Schulmeyer, *et al*, The Handbook of *Software* Quality Assurance (Upper Saddle River, NJ: Prentice Hall PTR, 1998), p. 9.

A garantia da qualidade de *software* é um conjunto de atividades relacionadas aplicadas através do ciclo de vida do *software* para influenciar positivamente e quantificar a qualidade do *software* entregue. Segundo descreve o relatório técnico CRSIP (2000) ressalta que a garantia da qualidade de *software* não está exclusivamente associada com as atividades principais do desenvolvimento do *software*, e sim, expande-se através de todo o ciclo de vida do mesmo.

O relatório menciona que a garantia da qualidade de *software* é composta pela garantia do processo e do produto, e, que, dentre seus métodos estão as atividades de *assessment*, funções de análises como predição da confiabilidade e análise causal, e, aplicação de métodos de detecção de defeitos como inspeções formais e testes (CRSIP, 2000, p. 2).

Controle da qualidade, pode ser definido, segundo a norma britânica BS 4778 (HUTCHESON, 2003, p. 25), como "técnicas e atividades operacionais que são utilizadas para satisfazer os requisitos de qualidade", ou, como (LEWIS, 2000, p. 9) "... o processo pelo qual a qualidade do produto é comparada com as normas aplicáveis, e a ação tomada quando uma não conformidade é detectada". Neste caso, trata-se de um esforço planejado para garantir que o *software* atenda estes critérios e possua atributos adicionais específicos para o projeto, como por exemplo, portabilidade, eficiência, reusabilidade, e flexibilidade. É uma coleção de atividades e funções utilizadas para monitorar e controlar um projeto de *software* de forma que objetivos específicos sejam atingidos com um nível desejado de confiança (LEWIS, 2000, p. 6).

Outra definição de Garantia da Qualidade de *Software* é (LEWIS, 2000, p. 6): "atividades sistemáticas que evidenciam a adequação ao uso do *software*".

Os objetivos da qualidade de *software* são tipicamente, segundo o autor Lewis (2000, p. 7), atingidos seguindo um plano de garantia de qualidade de *software* que estabelece os métodos que o projeto irá empregar para garantir que os documentos e produtos produzidos e revisados em cada *milestone* sejam de alta qualidade. O plano estabelece o critério pelo qual as atividades de qualidade podem ser monitoradas ao invés de trazer objetivos impossíveis, como por exemplo, *software* sem defeitos ou 100% confiável (LEWIS, 2000, p. 7).

Lewis (2000, p. 7) menciona ainda que a garantia da qualidade de *software* é uma estratégia para gerenciamento de risco. Ela existe porque a qualidade do *software* é geralmente custosa e deve ser incorporada no gerenciamento formal de riscos de um projeto. O autor traz alguns exemplos de situações decorrentes de *software* com pouca qualidade:

a) O *software* freqüentemente falha;
b) Conseqüências incaceitáveis da falha do *software*, desde financeiras até relacionadas à ameaças à vida;
c) Sistemas freqüentemente não estão disponíveis para a sua finalidade;
d) Melhorias do sistema são freqüentemente muito caras;
e) Os custos de detecção e remoção de defeitos são excessivos.

Lewis (2000, p. 7) observa ainda que algumas vezes os requisitos do sistema são inadequados ou mesmo incorretos. Como um defeito é uma falha em atender um requisito em sua visão, nestas situações os riscos de defeitos são maiores e mais críticos, pois o resultado é a existência de muitos defeitos na construção e produtos que não são verificáveis.

2.3.5 Elementos de um sistema de Qualidade

Diversos são os elementos que compõe os sistemas de qualidade: auditorias, os métodos de detecção de erros (*Embedded SQA Error Detection Methods*), *Assessment* e Análises (CRSIP, 2000, p. 2-4).

No conjunto dos métodos de detecção de erros estão a Inspeção Formal, Revisões, Walkthroughs e Testes (CRSIP, 2000, p. 2-4).

Em se tratando das Análises, o relatório CRSIP (2000, p. 3-4) apresenta os Processos de Análise Causal e Prevenção de defeito, Previsão de Confiabilidade e o Controle Estatístico de Processo.

Lewis (2000, p. 8) afirma que um sistema de garantia de qualidade de *software* é composto pelo teste de *software*, ou seja, a verificação e a validação, pelo gerenciamento de

configuração de *software* e pelo controle de qualidade dentro de uma coleção de normas, práticas, convenções e especificações, conforme mostrado na Figura 2.

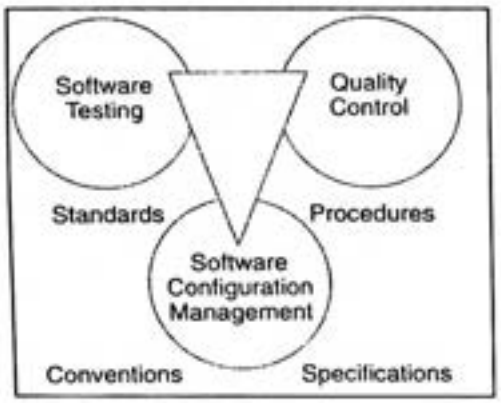

Figura 2 – Componentes da garantia de qualidade

FONTE: LEWIS, 2000, p. 8

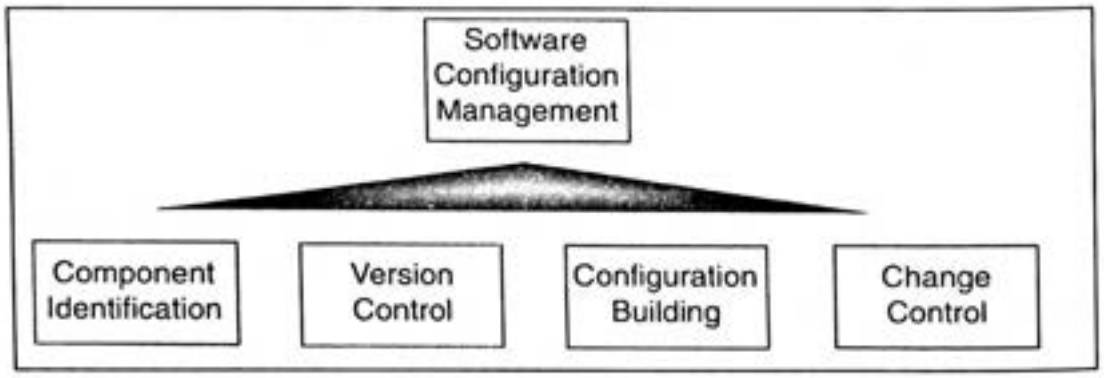

Figura 3 – Gerenciamento da configuração de *Software*

FONTE: LEWIS, 2000, p. 11

Alguns autores, como De Young e Rätzmann (2003, p. 25), dividem a garantia da qualidade em garantia da qualidade produtiva e garantia da qualidade analítica. Segundo os autores, a garantia da qualidade produtiva inclui todas as atividades que melhoram o produto, enquanto a garantia da qualidade analítica agrupa as atividades que verificam seus níveis de qualidade. Nesta classificação, por exemplo, o teste de *software* pertence à garantia da qualidade analítica.

Já Horch (2003) aponta os seguintes elementos como sendo os componentes de um sistema de qualidade de *software*:

- Normas (*standards*);
- Revisões (*reviewing*);
- Testes (*testing*);
- Análises de defeito (*defect analysis*);
- Gerenciamento de configuração (*configuration management - CM*);
- Segurança (*security*);
- Educação (*education*);
- Gerenciamento do *Vendor* (*vendor management*);
- Segurança (*safety*);
- Gerenciamento de Risco (*risk management*).

Horch (2003, p. 2) menciona que existem dois objetivos nestes sistemas - construir a qualidade no *software* desde o início e mantê-la ao longo de seu ciclo de vida – e que cada elemento contribui com um grau de intensidade diferente com cada um destes objetivos.

O autor adota a seguinte nomenclatura para as diferentes fases do ciclo de vida de *software* e as relaciona com cada um dos elementos do sistema de qualidade de *software* e com os objetivos na Figura 4 (HORCH, 2003, p. 2):

- Reconhecimento de uma necessidade ou problema (p.e., definição de conceito);
- Definição da solução de *software* a ser aplicada (p.e., definição de requisitos);
- Desenvolvimento do *software* que resolve o problema ou satisfaz as necessidades (p.e., modelagem e codificação);
- Prova de que a solução está correta (p.e., teste);
- Implementação da solução (p.e., instalação e aceite);
- Utilização da solução (p.e., operação);
- Melhoria da solução (p.e., manutenção);

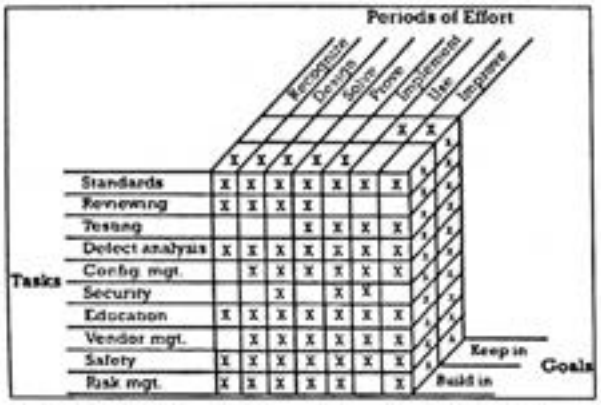

Figura 4 - Tarefas de Qualidade, períodos do ciclo de vida e objetivos

FONTE: HORCH, 2003, p. 2

Para Horch (2003, p. 2), visto que o *software* está se movendo de "uma arte arcaica para uma ciência visível" e que a liberdade da criatividade está perdendo espaço para abordagens mais controladas e científicas, as normas se fazem necessárias, pois objetivam prover métodos consistentes, rigorosos e uniformes para o desenvolvimento de *software*. A Figura 5 mostra as diferentes origens das normas segundo o autor.

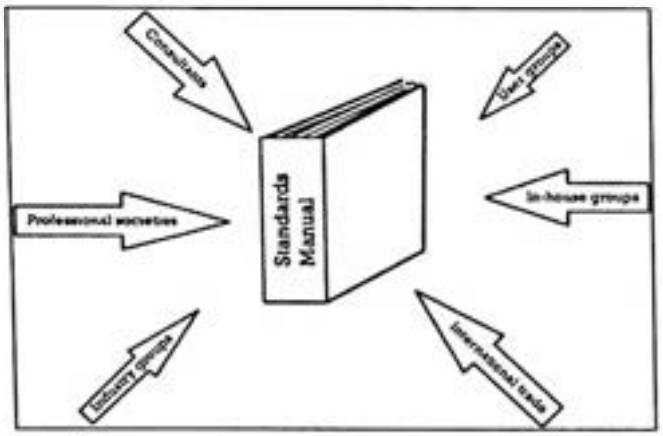

Figura 5 - Fontes das normas

FONTE: HORCH, 2003, p. 3

O autor comenta ainda que independentemente da origem, algumas características precisam estar presentes, tal como a necessidade, a exequibilidade, e a mensurabilidade, já que a ausência de uma destas características faz com que suas normas percam sua força e aplicabilidade (HORCH, 2003, p. 2-3). Além disto o autor descreve que em sua visão nem

tudo deve ser normalizado, sob a pena de perder a eficiência e a simplicidade, bem como prejudicar tarefas nas quais algum grau de liberdade se faz necessário (HORCH, 2003, p. 3).

As revisões são atividades que permitem uma visibilidade durante a execução do desenvolvimento de *software* e as atividades de instalação. As revisões de produto, também conhecidas como revisões técnicas, são exames formais e informais de produtos e componentes através das fases do desenvolvimento de *software* dentro de seu ciclo de vida. As revisões informais incluem as inspeções e os walk-throughs (HORCH, 2003, p. 3).

As revisões de processos examinam o sucesso do processo de desenvolvimento de *software* visando identificar oportunidades de melhorias no processo atual. A auditorias são exames de componentes para verificar a conformidade com especificações de formato e conteúdo (HORCH, 2003, p. 3-4).

Os testes visão prover uma confiança crescente e uma demostração de que os requisitos do *software* estão sendo satisfeitos, na visão do autor Horch (2003, p. 4). As atividades de teste incluem planejamento, modelagem, execução e *reporting*.

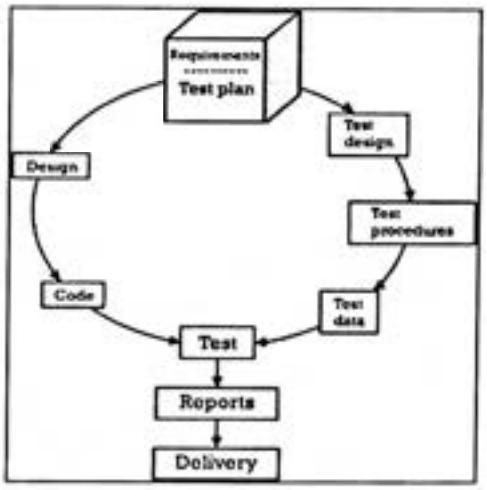

Figura 6 - Processo de teste simplificado
FONTE: HORCH, 2003, p. 4

Analises de defeito é a combinação de detecção de defeitos e correções, bem como analises de tendências de defeito.

Os esforços de análise de tendência de defeitos provem o armazenamento da ocorrência dos defeitos, suas soluções bem como seus *status*. Isto pode servir para (HORCH, 2003, p. 5):

- prevenção da permanência de defeitos não resolvidos por longos períodos de tempo;
- prevenção de mudanças não garantidas;
- apontamentos de áreas "fracas" ou vulneráveis no *software*;
- provimento de dados de análise para a avaliação e a correção do processo de desenvolvimento;
- provimento de avisos de potenciais defeitos através da análise das tendências de defeito.

No entanto, o autor ressalta que este procedimento de armazenamento dos dados é insuficiente se os relatórios necessários não forem gerados e o gerente de projetos não tiver a visibilidade destes no contexto do progresso e status do projeto (HORCH, 2003, p.5).

O autor descreve a gerência de configuração como uma disciplina composta por três "pastas" (Figura 7) – Identificação da Configuração, Controle da Configuração e Contabilidade da Configuração (HORCH, 2003, p.5).

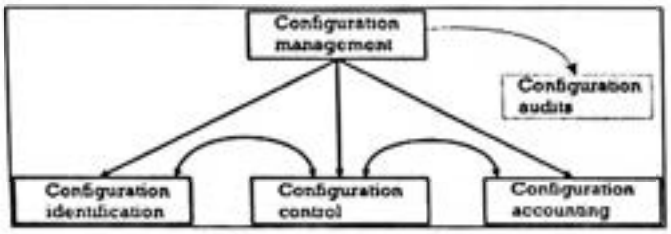

Figura 7 - Atividadades de Gerência de Mudanças (CM)
FONTE: HORCH, 2003, p. 5

As atividades de segurança (*security*) são aplicáveis tanto para acesso lógico quanto físico aos dados. Estas atividades visam proteger a disponibilidade do *software* e de seus ambientes. Por maior que seja a qualidade do *software*, lembra o autor Horch(2003), não existe utilidade no mesmo seu banco de informações estiver danificado ou destruído, seja por incêndio, ataques terroristas ou modificações desconhecidas ou não autorizadas, feita intencionalmente ou por acidente. Outra preocupação importante é o acesso e roubo de informações por *hackers* (HORCH, 2003, p.7).

A educação garante que as pessoas envolvidas no desenvolvimento de *software*, e aquelas utilizando o *software*, uma vez que ele esteja desenvolvido, estão aptas a executar suas tarefas corretamente (HORCH, 2003, p.7).

O gerenciamento do *vendor* se deve ao fato que, de acordo com o seu tipo de compra, cada *software* terá sua própria abordagem de sistema de qualidade e cada um deve ser suportado de uma forma apropriada de acordo com o grau de controle que o comprador tem sobre o processo de desenvolvimento utilizado pelo produtor. Os tres tipos básicos de compra de *software* são (HORCH, 2003, p.8):

a) Prateleira (*Off-the-shelf*);
b) Adaptado (*Tailored shelf*);
c) Contratado.

O *software* de prateleira é um pacote que compramos em uma loja, como por exemplo um editor de textos, um anti-virus, um sistema operacional, etc. Estes pacotes não vem com garantia de que os mesmos irão fazer aquilo que o cliente precisa, ou seja, eles estão quase que totalmente fora da influência do comprador no que diz respeito a qualidade (HORCH, 2003, p.8).

A segunda categoria trata de situações em que um pacote ou *framework* é comprado e o *vendor* adiciona algumas capacidades específicas requeridas pelo contratante. A esfera de influência de qualidade está apenas sobre as partes customizadas. O exemplo desta situação são os pacotes de gestão, ERPs, CRMs, dentre outros (HORCH, 2003, p.8).

A terceira categoria é o *software* contratado. Nesta situação ele é contratualmente especificado e, neste caso, portanto, a esfera de influência do comprador sobre o produtor é grande (HORCH, 2003, p.8).

O autor ressalta que a atenção às práticas de qualidade do *vendor* são extremamente importantes em situações que o desenvolvimento é *off-shore*, como na Índia ou Malásia, por exemplo (HORCH, 2003, p.8).

Cada projeto de *software* deve conscientemente considerar as implicações de segurança (safety) do mesmo e em relação ao sistema do qual ele fará parte visto que a importância do mesmo e os impactos são cada vez maiores em nossas vidas, pois suas aplicações incluem equipamentos médicos, controle de transportes, dentre outros. Segundo o Horch (2003, p. 9), o plano de gerenciamento de projeto deve incluir um parágrado descrevendo os "issues" de segurança a serem considerados. Se for o caso, o autor menciona ainda que um plano de segurança do *software* deve ser preparado (HORCH, 2003, p.9)

O gerenciamento do risco inclui a identificação do risco, a determinação da probabilidade, o custo e a ameaça do risco, bem como as ações que devem ser tomadas para eliminá-lo, reduzi-lo ou mesmo aceitá-lo (HORCH, 2003, p.9).

2.3.6 Ferramentas para o controle de qualidade

Diversas são as ferramentas utilizadas para o controle de qualidade. Horch (2003, p.115) menciona que as medidas e métricas podem tomar diversas formas. As ferramentas mais freqüentes, segundo o autor, para apresentar as medidas e métricas coletadas são (HORCH, 2003, p. 115-123):

- Folha de registro (*Tally sheet*);
- Diagrama de dispersão (*Scatter diagram*);
- *Graph*;
- Histogramas;
- Diagrama de Pareto;
- Fluxogramas;
- Diagrama de causa e efeito ou espinha de peixe (*Fishbone*);
- Carta de controle estatístico (*Statistical control chart*).

A folha de registro é a forma mais simples de coleção de dados. Cada ocorrência de algum evento é registrada da forma como ocorreu ou foi detectada. Trata-se apenas de uma coleção de defeitos detectados (Figura 8)(HORCH, 2003, p. 115).

Module	Defect count (× five)	Total
1	||||	20
2	𝍸 |	30
3	𝍸 𝍸 Plus 4	54
4	𝍸 Plus 2	27
5	𝍸 |||	40
6	𝍸 𝍸 Plus 2	52
7	𝍸 𝍸 𝍸 ||	80
8	𝍸 ||	35
9	𝍸 || Plus 3	43
10	|| Plus 2	12
11	𝍸 | Plus 1	31
12	𝍸 𝍸 Plus 3	53

Figura 8 – Folha de Registro

FONTE: HORCH, 2003, p. 115

O gráfico de dispersão é uma representação gráfica dos dados da folha de registro de uma forma mais matemática, e que permite uma comparação visual dos números. Às vezes utiliza-se uma linha de tendência ou a curva dos mínimos quadrados para sumarizar os pontos dispersos. Estas linhas representam uma estimativa da tendência dos dados (Figura 9)(HORCH, 2003, p. 115).

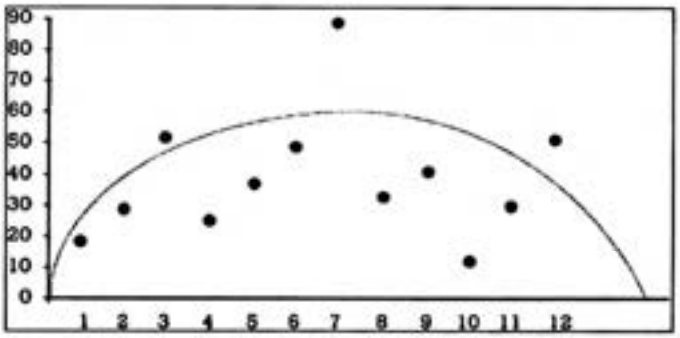

Figura 9 - Diagrama de dispersão

FONTE: HORCH, 2003, p. 116

Um *graph* é um gráfico de dispersão com os pontos conectados, de uma forma simplista. Esta forma é preferida para observação do progresso de um processo ou atividade em relação ao tempo. Mais informações, como datas ou complexidade, podem ser mostradas no gráfico (Figura 10) (HORCH, 2003, p. 116).

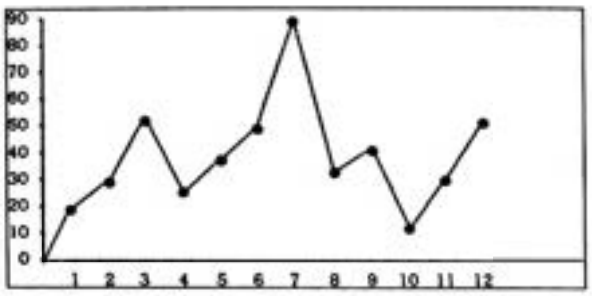

Figura 10 – *Graph*
FONTE: HORCH, 2003, p. 116

O histograma é similar ao *graph*, no entanto, ao invés de conctar os pontos com uma linha, eles são mostrados como barras horizontais ou verticais. Esta ferramenta é a precursora dos diagramas de Pareto e, algumas vezes, é expandida de maneira que a largura das barras represente condições adicionais, como tamanho ou esforço, por exemplo (Figura 11)(HORCH, 2003, p. 116).

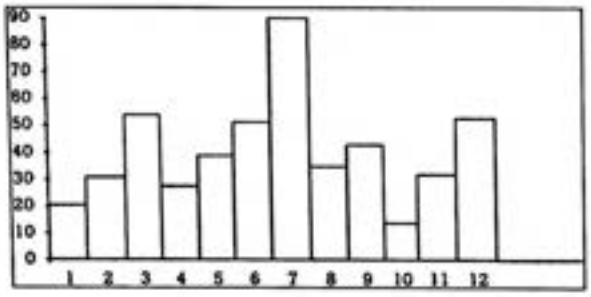

Figura 11 - Histograma
FONTE: HORCH, 2003, p. 117

A regra 80/20, criada pelo economisma Vilfredo Pareto no séc. XIX (Horch, 2003) e aplicado por Juran em 1950, para a identificação dos problemas de qualidade, sugere que deve-se prestar atenção à porção que detenha em torno de 80% dos defeitos, já que a maior parte dos problemas de qualidade são devidos a um pequeno percentual de possíveis causas (KAN, 2003, p. 133).

O diagrama de Pareto é o histograma arranjado de forma decrescente em termos da altura das barras. A idéia é utilizar este diagrama para priorizar os exames das causas dos npumeros de defeitos associados a cada módulo (Figura 12)(HORCH, 2003, p. 117).

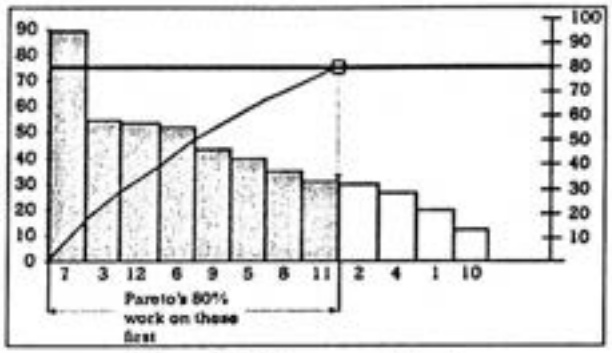

Figura 12 - Diagrama de Pareto
FONTE: HORCH, 2003, p. 117

Os fluxogramas são diagramas que permitem a descrição de processos. Eles não são utilizados para representar dados. Na garantia da qualidade de *software*, segundo o autor, eles são utilizados para representar os vários processos utilizados no desenvolvimento do *software*, na manutenção, em sua operação e na melhoria. Quando as métricas começam a sugerir falhar em um ou outro processo, o fluxograma pode ajudar a isolar a parte do processo que está com problemas (Figura 13)(HORCH, 2003, p. 118).

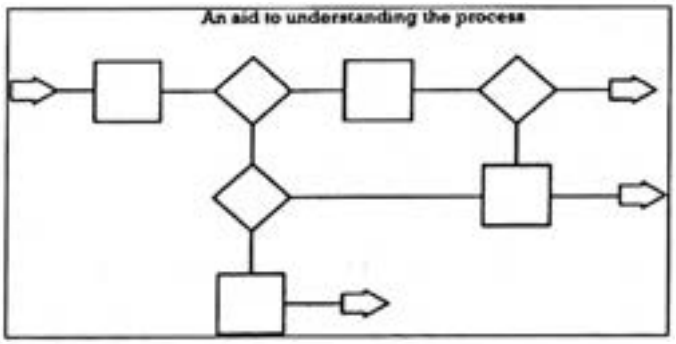

Figura 13 - Fluxograma
FONTE: HORCH, 2003, p. 119

Um outro diagrama utilizado para localizar falhas em processos é o de causa e efeito. Estes diagramas são utilizados para a determinação da causa de um defeito, de forma que não apenas o defeito seja eliminado, mas também a situação que permitiu sua introdução seja também diagnosticada e eliminada, de forma que não ocorra novamente. Este digrama é também conhecido como diagrama de Ishikawa (Figura 14)(HORCH, 2003, p. 119).

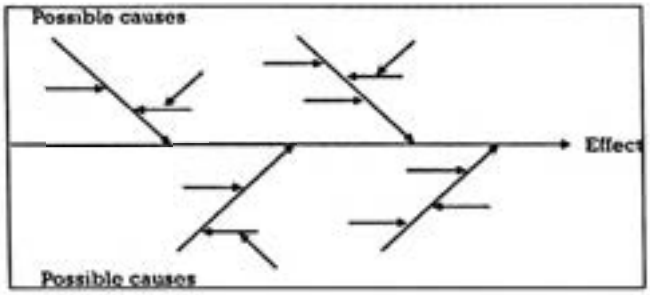

Figura 14 – Diagrama de causa e efeito
FONTE: HORCH, 2003, p. 119

Horch (2003, p. 119-120) apresenta ainda os diagramas de controle de processos criados em 1931 por Walter Shewhart como ferramentas de apoio ao controle da qualidade do *software* – Carta de funcionamento (Run charts) e Carta de controle de aceite (Acceptance control charts).

As cartas de funcionamento descrevem o comportamento estatístico de um processo. Shewhart pretendeu que elas fossem usados para mostrar quando um processo está estável ou sob controle. Não é baseado no comportamento pretendido ou alvo do processo mas de seu comportamento real. A Figura 15 mostra uma carta básica do controle do processo. A linha central é o nível médio da característica descrita (talvez a taxa de erro para um sistema de *software* dado). A linha superior é o limite de controle superior (UCL) e indica a presença de causas especiais dos defeitos. A linha mais baixa é o limite de controle mais baixo (LCL) e reflete também causas especiais, no entanto porque o produto é melhor do que ele tem que ser. Shewhart ajustou o UCL e o LCL como funções do desvio médio e padrão para a população que está sendo traçada (HORCH, 2003, p. 120).

Figura 15 – Carta de funcionamento básica
FONTE: HORCH, 2003, p. 121

A idéia de avaliar o comportamento do processo utilizando uma carta de funcionamento é descrita na Figura 16. Um processo que tenha uma taxa de erro caindo consistentemente entre os limites de controle (como na primeira seção da Figura 16) é dito estar no controle. Um ponto ocasional fora dos limites identifica um argumento especial, um alvo para a causa e a análise do efeito. No general, entretanto, o processo é considerado sadio e repetível. A segunda seção da Figura 16 mostra o caso em que os pontos caem aleatóriamente em torno do meio. Este processo é dito estar fora do controle. As mudanças no processo não são possíveis de serem identificadas com mudanças nos resultados. A terceira seção da Figura 16 mostra um processo no controle mas a caminho de sair deste. No mundo da manufatura, poderia-se concluir, segundo Horch (2003, p. 21), que uma ferramenta estava ficando desgastada. No *software* pode-se suspeitar que uma técnica de deteção do defeito está melhorando (a tendência implica que estão sendo encontrados mais defeitos) ou que o processo do desenvolvimento está degenerando (e deixando mais defeitos nos produtos) (HORCH, 2003, p. 121).

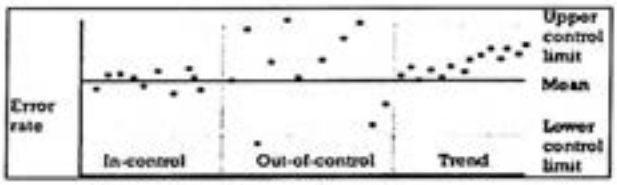

Figura 16 – Comportamento de processo
FONTE: HORCH, 2003, p. 121

A melhoria contínua de processo pode ser descrita como na Figura 17. Baseado no conceito japonês de *kaizen* (melhoria contínua através das mudanças pequenas), o processo é melhorado continuamente. Em conseqüência, os limites de controle, funções do desvio médio e padrão, tendem a mover-se para mais próximo ao meio (HORCH, 2003, p. 121-122).

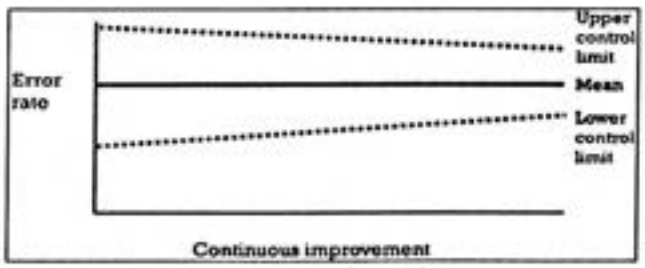

Figura 17 – Conceito *Kaizen*

FONTE: HORCH, 2003, p. 122

As cartas de controle de aceitação não são baseadas na determinação estatística dos limites do meio e de controle. Estas cartas usam o valor desejado, chamado de alvo, como a linha central. Os limites de controle superiores e inferiores são escolhidos baseando-se na variação permissível e não na variação estatística (HORCH, 2003, p. 122).

Em termos do *software*, ao descrever defeitos, deseja-se que todas as três linhas encontrem-se na linha do zero-defeito. Este é também o objetivo final a longo prazo final do controle da qualidade de *software*, segundo o autor. No entanto, no mundo real, existem outras pressões que tornam mais lenta a realização do objetivo de zero-defeito. Em termos econômicos, pode-se dizer que o UCL é o nível em que o cliente não aceita mais o produto, e o LCL o nível em que os custos de encontrar uns defeitos adicionais excedem os custos incorridos se o defeito ocorrer no uso do *software*. Indicado ainda de uma outra maneira, o UCL define "o quão ruim ousa-se fazer" e o LCL define "quão bom é possível com recursos disponíveis." (HORCH, 2003, p. 122-123).

As cartas de controle de aceitação não são tão estatisticamente válidas quanto as cartas de funcionamento, porém não requerem os grandes tamanhos da amostra e da população que as cartas de funcionamento geralmente demandam. Ainda, segundo o autor, elas são adaptadas mais facilmente aos usos do profissional da garantia de qualidade do *software*. A Figura 18 combina a carta de controle da aceitação, o conceito *kaizen*, o desejo para o defeito-zero, e a realidade econômica, em uma carta típica do controle do processo de desenvolvimento de *software*. Nesta carta, o LCL é ajustado como perto de zero da forma que é economicamente praticável. A linha central, a taxa alvo de defeito, começa no valor experimentado atualmente e inclina-se para baixo em direção a taxa objetivo do sistema de garantia da qualidade de *software*. Ao mesmo tempo, o UCL é inclinado mais acentuamente para motivar a melhoria de processos (HORCH, 2003, p. 123).

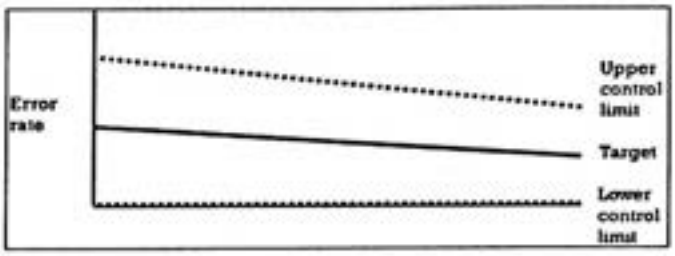

Figura 18 – Rumo ao defeito zero
FONTE: HORCH, 2003, p. 123

Kan (2003, p. 128) menciona também a utilização das ferramentas estatísticas básicas de controle de qualidade de Ishikawa (Figura 19), amplamente utilizadas na produção de manufaturados. No entanto, ele ressalva que o processo de desenvolvimento de *software* é complexo e envolve um elevado grau de criatividade e atividade intelectual, e que, por este motivo, fica muito difícil, se não impossível, definir a capacidade de processo do desenvolvimento de *software* em termos estatísticos, de forma que o controle estatísticos de processo nesta atividade vai além do controle por diagramas, exigindo, por exemplo, novas tecnologias, ferramentas CASE ((R)), modelos de defeitos e técnicas de estimativa da confiabilidade. Por outro lado, o uso das sete ferramentas básicas, segundo o autor, pode trazer resultados positivos no longo prazo em termos de melhoria de processo e gerenciamento da qualidade no desenvolvimento de *software* (KAN, 2003, p. 128).

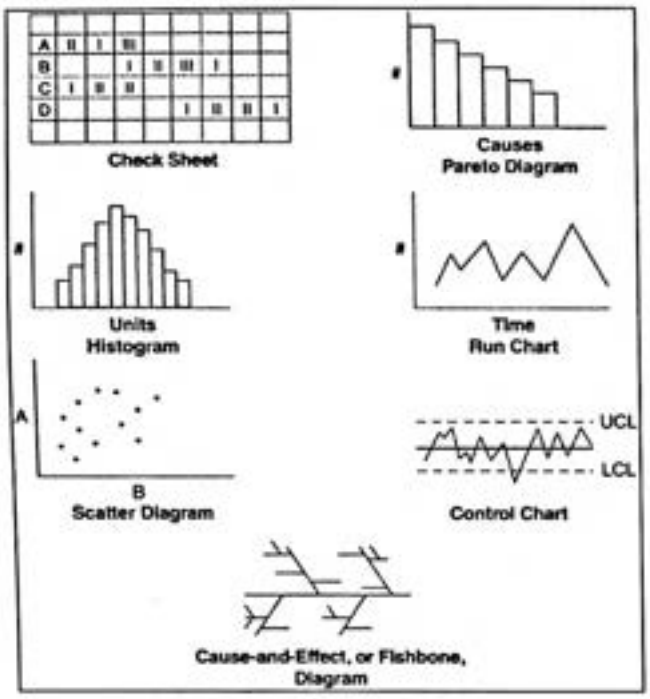

Figura 19 - As sete ferramentas básicas

FONTE: KAN, 2003, p. 129

2.3.7 Modelos da Qualidade

A literatura no assunto cobre diversos modelos da qualidade de produto, os mais conhecidos na ordem cronológica são: Modelo de (1977), Modelo de Boehm (1978), FURPS (1987), ISO 9126 (1991) e Modelo de Dromey (1996) e o Modelo Sistêmico de Qualidade (Systemic Quality Model). (PÉREZ *et al*, 2004)

Analisando-se estes modelos da qualidade, é possível identificar vantagens e desvantagens destes modelos ao considerar o modelo da qualidade do *software* com uma abordagem sistêmica. (PÉREZ *et al*, 2004)

Uma das principais contribuições do modelo de McCall é o relacionamento criado entre características da qualidade e métricas, embora existam críticas que nem todas as métricas sejam objetivas. Um aspecto não considerado diretamente por este modelo era a funcionalidade do produto de *software*. O modelo de Boehm é similar ao modelo de McCall

no fato que ele representa uma estrutura hierárquica das características, cada uma delas contribuindo para a qualidade total. Boehm inclui as necessidades dos usuários, como McCall. Entretanto, ele adiciona também as características do rendimento do hardware não encontradas no modelo de McCall. Uma desvantagem do modelo de FURPS é que ele falha em não levar em conta a portabilidade do produto *software*. Uma das vantagens do modelo do ISO 9126 é que ele identifica as características internas e características externas da qualidade de um produto de *software*. Entretanto, ao mesmo tempo tem a desvantagem de não mostrar muito claramente como estes aspectos podem ser medidos (PÉREZ *et al*, 2004).

O modelo de Dromey procura aumentar a compreensão do relacionamento entre os atributos (características) e os sub-attributos (sub-características) da qualidade. Tenta também localizar as propriedades do produto de *software* que afetam os atributos da qualidade. As características da qualidade encontradas na maioria dos modelos são: eficiência, confiabilidade, manutenabilidade (maintainability), portabilidade (portability), usabilidade e funcionalidade. Pelo fato de estarem presentes em todos os modelos estudados, podem ser considerados essenciais e dignas do estudo (PÉREZ *et al*, 2004).

Outros estudos que suportam a idéia da qualidade sistêmica, levando em conta a qualidade do produto e do processo, foram apresentados por Dromey e por Voas. Dromey diz que a qualidade de processo não pode existir se não estiver baseada em um modelo de qualidade de produto (PÉREZ *et al*, 2004).

Um modelo que difere ligeiramente dos demais citados, já que não apresenta nenhuma estrutura hierárquica, é o modelo de Callaos. Há uma similaridade entre os conceitos usados por Callaos, na forma como considera eficiência e eficácia do produto, e as definições das características internas e externas do produto (PÉREZ *et al*, 2004).

Callaos quer dizer que deve haver um balanço entre a eficiência e a eficácia, o que nos conduz a concluir que um cuidado especial deve ser tomado com as características internas e externas do produto. (PÉREZ *et al*, 2004)

2.3.8 Fatores de Qualidade

Diversos atributos para a qualidade de *software* são encontrados na literatura especializada. Juran chama estes atributos de parâmetros da qualidade ou parâmetros de adequação ao uso (KAN, 2003, p. 5). Para outros autores, como em Tsao *et al* (2003), o termo utilizado para *design*ar tais atributos é "fatores da qualidade".

Tsao *et al* (2003, p. 312-316) apresenta sua seleção de fatores da qualidade baseado no trabalho de J. A. McCall *et al*. No entanto, em seu trabalho, Tsao *et al* (2003, p. 312-316) incluiu algumas melhorias baseadas na ISO/IEC 9126-1—Standard for *Software* Engineering—Product Quality—Part 1: Quality Model.

A Tabela 8 lista os fatores de qualidade dentro da perspectiva do usuário de um componente de *software*, bem como a definição adotada em Tsao *et al* (2003, p. 312-313).

Tabela 8 – Fatores da qualidade: perspectiva do usuário de um componente de *Software*

Fator de Qualidade	Definição
Funcionalidade (Functionality)	A extensão com que um componente satisfaz a suas especificações e cumpre as necessidades indicadas ou implícitas do usuário.
Eficiência (Efficiency)	A quantidade de recurso computacional requerida por um componente para cumprir suas funções sob dadas circunstâncias.
Confiabilidade (Reliability)	A extensão com que se pode esperar que um componente cumpra suas funções para um dado período de tempo e certas condições.
Segurança (Integridade) / Security (Integrity)	A extensão com que o acesso por pessoas não autorizadas a um componente de *software*, a um *software* baseado no componente em questão ou os dados do componente pode ser controlado.
Usabilidade (Usability)	A extensão da facilidade a que um componente de *software* pode ser desempacotado por uma variedade dos usuários, configurado por estes para melhor satisfazerem às suas necessidades, e montado por estes usuários nos ambientes da aplicação de seus sistemas. Inclui também a compreensibilidade e a facilidade da aprendizagem.
Portabilidade (Portability)	A extensão a que um componente de *software* pode ser portado a uma possivelmente ampla variedade de ambientes operacionais, incluindo sistemas operacionais e hardwares, e a quantidade de esforço requerido para portá-lo.
Interoperabilidade (Interoperability)	A extensão a que um componente de *software* pode ser utilizado em uma ampla variedade de sistemas de *software*.
Verificabilidade (Verifiability)	O esforço requerido para verificar, com ou sem acesso ao código de fonte, à arquitetura, ao projeto e aos colaboradores, que o projeto do *software* e a execução satisfazem às

	especificações do componente de *software*. O autor comenta que a Verificabilidade vai além da testabilidade, que se refere ao esforço necessário para se assegurar de que o componente execute função desejada com o desempenho pretendido. A testabilidade é um fator de qualidade geralmente aceito [10,11]. A verificabilidade pode incluir o conceito de certificabilidade, que pode ser definido como a extensão com que um componente de *software* pode ser certificado, a extensão com que a certificação implica a qualidade do componente de *software*, e o custo de tais certificações.
Manutenabilidade (Maintainability)	O esforço requerido substituir um componente de *software* por uma versão corrigida, atualizar um componente de *software* atual, e migrar um componente de *software* existente de um sistema de *software* atual para uma nova versão do sistema.

FONTE: TSAO *et al*, 2003, p. 312-313

Tsao *et al* (2003, p. 312-316) menciona que a comunidade de Garantia da Qualidade para *Software* têm utilizado uma hierarquia de qualidade: fatores de qualidade, critérios de qualidade e métricas de qualidade. Isto devido ao fato dos fatores de qualidade terem sido refinados em um conjunto maior de critérios mais independentes e específicos de qualidade. Dentre estes critérios comumente aceitos estão: rastreabilidade (*traceability*), completude (*completeness*), consistência (*consistency*), acurácia (*accuracy*), tolerância a erro (*error tolerance*), simplicidade (*simplicity*), modularidade (*modularity*), generalidade (*generality*), expansibilidade (*expandability*), instrumentação (*instrumentation*), auto-descritibilidade (*self-descriptiveness*), eficiência de execução (*execution* efficiency), eficiência de armazenamento (*storage efficiency*), controle de acesso (*access control*), auditoria de acesso (*access audit*), operabilidade (*operability*), treinamento (*training*), comunicabilidade (*communicativeness*), independência do sistema de *software* (*software systems independence*), independência da máquina (*machine independence*), concordância nas comunicações (*communications commonality*), concordância nos dados (*data* commonality), e concisão.

Tsao *et al* (2003, p. 312-316) apresenta ainda os relacionamentos entre estes critérios e os fatores de qualidade mais críticos, em sua visão, como: reusabilidade, usabilidade, portabilidade, interoperabilidade, funcionalidade (corretude), verificabilidade, e manutenabilidade. Os relacionamentos estão mostrados na Tabela 9.

Tabela 9 – Critérios de qualidade relacionados com fatores críticos da qualidade do componente

Fator de Qualidade	Definição
Funcionalidade (Functionality)	Traceability, consistency, completeness
Reusabilidade	Generality, modularity, *software* system independence, machine independence, self-descriptiveness (including document availability, accessibility, accuracy, completeness, and clarity)
Interoperabilidade	Modularity, communications commonality (standards compliance), data commonality, machine independence, *software* system independence, system compatibility
Portabilidade	Modularity, machine independence, *software* system independence, self-descriptiveness
Usabilidade	Operability, training, communicativeness
Verificabilidade	Simplicity, modularity, self-descriptiveness
Manutenabilidade	Consistency, simplicity, conciseness, modularity, self-descriptiveness

FONTE: TSAO *et al*, 2003, p. 316

A IBM utiliza um conjunto específico de fatores de qualidade do produto *software* definidos pelo CUPRIMDSO (capacidade [funcionalidade], usabilidade, performance, confiabilidade [*reliability*], instalabilidade, manutenabilidade, documentação/informação, serviço e outros), enquanto a Hewlett-Packard foca no FURPS (funcionalidade, usabilidade, confiabilidade, performance, capacidade de serviço). Outras empresas utilizam dimensões similares. (KAN, 2003, p. 4)

Estes atributos de qualidade nem sempre são congruentes entre si (KAN, 2003, p. 5). Kan (2003, p. 5) descreve a situação de quanto maior a complexidade funcional, mais difícil é obter-se a manutenabilidade do sistema. Em situações de transações on-line, como no caso de bancos por exemplo, a performance e a confiabilidade podem ser os atributos mais importantes. Para clientes com sistemas *standalone* ((R)) ou *client/server* ((R)) e com operações simples, a usabilidade, a instalabilidade e a documentação, podem por exemplo ser mais importantes.

A Figura 20 mostra os possíveis relacionamentos entre alguns atributos de qualidade de acordo com Kan (2003, p. 5) para o conjunto específico da IBM. Alguns relacionamentos são mutuamente suportados, outro são excludentes, e alguns outros não tão claros, pois dependem dos tipos de clientes e aplicações em questão. Kan (2003, p.5) ressalta que em casos de

software com um conjunto amplo de clientes, no entanto, setar objetivos para diversos atributos de qualidade e atender os requisitos dos clientes não é uma tarefa fácil.

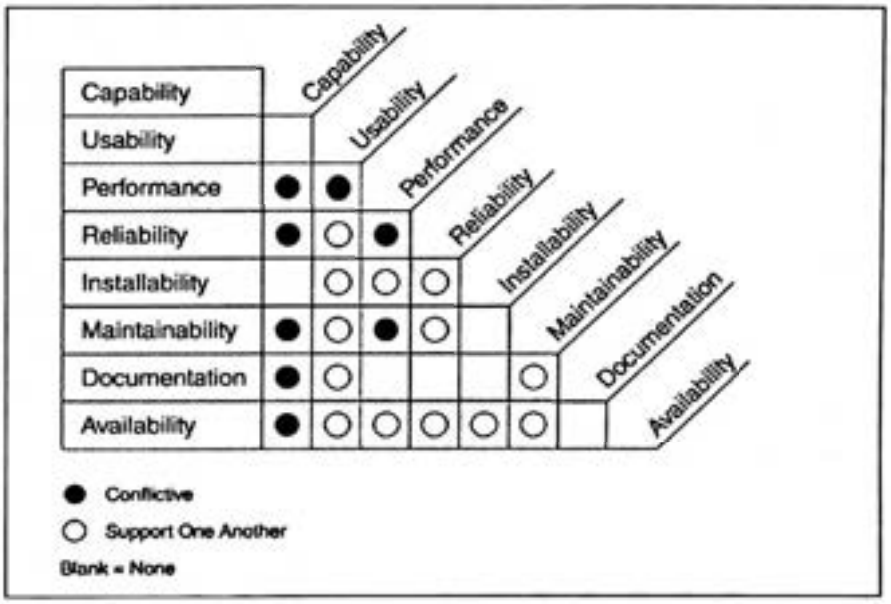

Figura 20 - Inter-relacionamentos de atributos de *software* - Exemplo com CUPRIMDA
FONTE: KAN, 2003, p.5

2.4 Processos de desenvolvimento do *software*

As mudanças são cada vez mais constantes, principalmente em termos de desenvolvimento de *software*, para atender as necessidades dos usuários. Isto afeta a maneira como os *softwares* são desenvolvidos. Em um ambiente sob constante mudança, principalmente na era da internet, todo tipo de burocracia relacionada ao desenvolvimento de *software* está sendo questionada e revisada (DE YOUNG; RÄTZMANN, 2003, p. 13), pois os processos devem ser flexíveis o bastante para acomodar tais mudanças.

Lewis (2000) reforça que, além das avaliações do produto, as avaliações do processo de desenvolvimento de software são essenciais para um programa de gerência da qualidade.

Lewis (2000) afirma que a qualidade não pode ser atingida avaliando-se um produto completo. A idéia, segundo o autor, é em primeiro lugar impedir os defeitos ou deficiências de qualidade e fazer com que os produtos sejam avaliáveis por medidas de garantia da qualidade. Algumas medidas da garantia de qualidade incluem: estruturação do processo do desenvolvimento com uma norma de *software* e o suporte ao processo do desenvolvimento com métodos, técnicas, e ferramentas.

Cada abordagem possui um foco diferente no momento em que os testes são realizados. Isto reflete não apenas na qualidade do produto ao final do desenvolvimento, como nos custos de mudança, tal como mostra a Figura 21.

Como as abordagens para a garantia da qualidade de *software* possuem forte relação com o modelo de desenvolvimento de *software* adotado, esta sessão traz uma revisão da literatura focando os principais modelos existentes e utilizados.

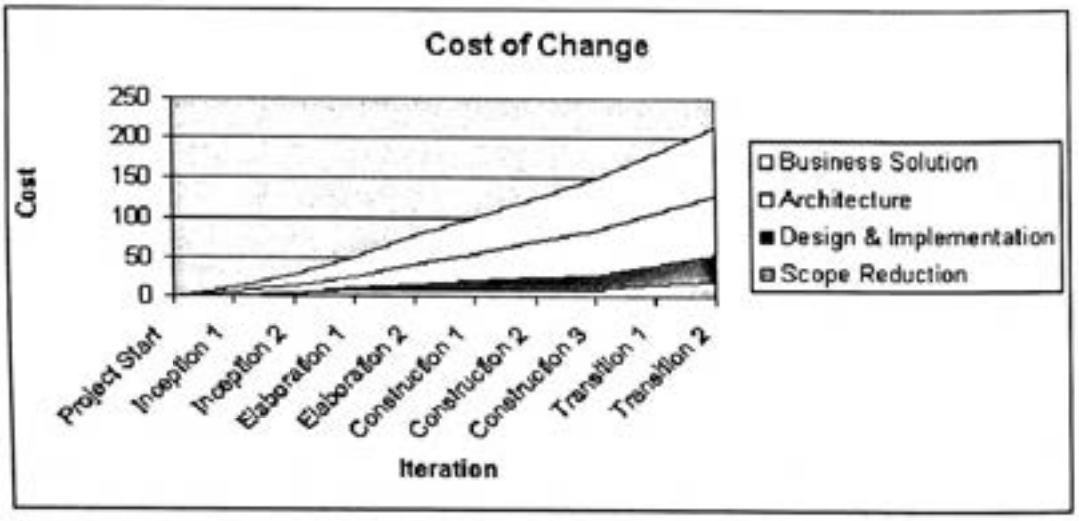

Figura 21 - Custo de mudanças ao longo das etapas de desenvolvimento
FONTE: Rational Edge, dezembro 2001

2.4.1 O modelo em cascata ou *waterfall*

Segundo Basili e Musa (1991), citados por Kan (2003, p. 14), nos anos 60 e 70, os projetos de desenvolvimento de *software* eram caracterizados por macissos estouros de orçamento e

atrasos. Nesta época o foco estava no planejamento e controle. Boehm (1976), citado por Kan (2003, p. 14) afirma que o surgimento do processo *waterfall*, para ajudar a lidar com a crescente complexidade do desenvolvimento de projetos, foi um acontecimento lógico (KAN, 2003, p. 14).

A Figura 22 mostra o processo *waterfall*. Este processo encoraja o time de desenvolvimento a especificar o que o *software* deve fazer (obter e definir os requisitos de sistema) antes de desenvolvê-lo. Sua idéia é quebrar a missão de desenvolver o *software* em algumas atividades lógicas (desenho, codificação, e assim por diante) com produtos intermediários que levam ao produto final. Para garantir a execução adequada e a boa qualidade dos produtos entregues, cada passo tem critérios de validação, entrada e saída. Este paradigma Entrada-Tarefa-Validação-Saída (Entry-Task-Validation-Exit (ETVX)) é a característica chave do processo *waterfall* de acordo com Radice *et al*, (1985) citado por Kan (2003, p. 19) (KAN, 2003, p. 14-19).

Kan (2003, p.14) afirma que a estratégia de dividir para conquistar empregada pela abordagem traz muitas vantagens como:

- permite o acompanhamento mais preciso do progresso do projeto e a identificação antecipada de possíveis problemas;
- força a organização que desenvolve o sistema de *software* para ser mais estruturada e gerenciável;
- esta abordagem estrutural é muito importante para organizações grandes com projetos de desenvolvimento grandes, complexos, pois exige que o processo gere uma série de documentos que podem ser usados depois para testar e manter o sistema (Davis *et al*, 1988) (apud Kan, 2003).

O objetivo desta abordagem, em linhas gerais, é tornar grandes projetos de *software* mais gerenciáveis e entregues no prazo sem custos adicionais. (KAN, 2003, p. 14)

Embora uma variedade de nomes seja dada a cada estágio no modelo, as metodologias básicas permanecem mais ou menos mesmas. Assim, os estágios de requisitos do sistema são chamados às vezes análise do sistema, recolhimento das exigências do cliente e análise, ou análise das necessidades do usuário; o estágio de modelagem pode dividido em modelagem de

alto nível e modelagem no nível de detalhe; o estágio da implementação pode ser chamado codificação e eliminação de erros (*debug*); e o estágio de testes pode incluir o teste a nível de componente, a nível de produto e a nível de sistema. (KAN, 2003, p.14-16)

Figura 22 mostra uma implementação do modelo do processo *waterfall* para um projeto grande. O autor chama a atenção ao fato de logo após o estágio de requisitos existir uma etapa de projeto arquitetural. Após estes estágios, o desgin e o desenvolvimento iniciam o trabalho de desenvolvimento de cada função. Isto consiste na modelagem (*design*) alto-nível (HLD), na modelagem (*design*) de baixo-nível (LLD), no desenvolvimento do código, e no teste unitário (UT). Apesar do conceito de *waterfall*, o paralelismo existe porque as várias funções podem ser desenvolvidas simultaneamente. Como mostrado na figura, o desenvolvimento do código e os estágios do teste de unidade são executados também de forma iterativa. Desde que UT é uma parte integral do estágio da execução, faz o algum sentido separá-lo em um outro estágio formal. Antes da conclusão do HLD, LLD, e o código, as revisões formais e as inspeções ocorrem como a parte da validação e critério de saída. Estas inspeções são chamadas I0, inspeções I1, e I2, respectivamente. Quando o código é terminado e a unidade está testada, os estágios subseqüentes são integração, teste componente, teste do sistema, e programas de teste envolvendo os cliente (early customer programs) – Testes Beta. O estágio final é liberação do sistema de *software* aos clientes. (KAN, 2003, p. 14-19)

Figura 22 - Um exemplo do modelo de processo em cascata (*Waterfall*)

FONTE: KAN, 2003, p.15

2.4.2 O Modelo em Espiral

O modelo espiral do desenvolvimento e melhoria do *software*, desenvolvido por Boehm (1988), é baseado na experiência com vários refinamentos do modelo *waterfall* para grandes projetos de *software* do governo. Confiando bastante na gerência de risco e na prototipação, é um modelo muito mais flexível do que o modelo da *waterfall*. A aplicação mais detalhada do modelo é o desenvolvimento do TRW - Sistema de produtividade de *software* (TRW-SPS) como descrita por Boehm. O conceito espiral e o foco da gerência de risco ganharam a aceitação em engenharia de *software* e na gerência de projetos nos anos recentes. (KAN, 2003, p.21)

A Figura 23 mostra o modelo espiral de Boehm. O conceito subjacente do modelo é que cada parcela do produto e cada nível da elaboração envolve a mesma seqüência de passos (ciclo). Começando no centro da espiral, pode-se ver que cada fase do desenvolvimento (conceito de operação, dos requisitos de *software*, da modelagem (*design*) do produto, da modelagem detalhada, e da execução) envolve um ciclo da espiral. A dimensão radial na figura XX representa o custo cumulativo incorrido em realizar as etapas. A dimensão angular representa o progresso feito em terminar cada ciclo da espiral. Como indicado pelos quadrantes na figura, a primeira etapa de cada ciclo da espiral é identificar os objetivos da parcela do produto que está sendo elaborado, dos meios alternativos da execução desta parcela do produto, e das restrições impostas na aplicação das alternativas. A etapa seguinte é avaliar as alternativas de acordo com os objetivos e as restrições, identificar os riscos associados, e resolvê-los. A análise do risco e a aproximação orientada a risco, são conseqüentemente características chaves do modelo espiral, no contraste à abordagem orientada a documento do modelo *waterfall*. (KAN, 2003, p.22)

Nesta aproximação orientada a risco, a prototipação é uma ferramenta importante. Geralmente a prototipação é aplicada aos elementos do sistema ou alternativas que apresento riscos mais elevados. Os protótipos insatisfatórios podem ser afastados; quando um protótipo operacional é escolhido, a implmentação pode começar. Além da prototipação, o modelo espiral usa simulações, modelos, e "benchmarks" a fim alcançar a melhor alternativa. Finalmente, como indicado na ilustração, uma característica importante do modelo espiral, como em outros modelos, é que cada ciclo termina com uma revisão que envolve os membros chave ou organizações envolvidas com no produto (KAN, 2003, p. 23).

Para projetos do *software* com desenvolvimento incremental ou com os componentes a ser desenvolvidos por organizações separadas ou por indivíduos, uma série de ciclos espirais pode ser usada, um para cada incremento ou componente. Uma terceira dimensão podia ser adicionada a Figura 23 para representar melhor o modelo. (KAN, 2003, p. 23)

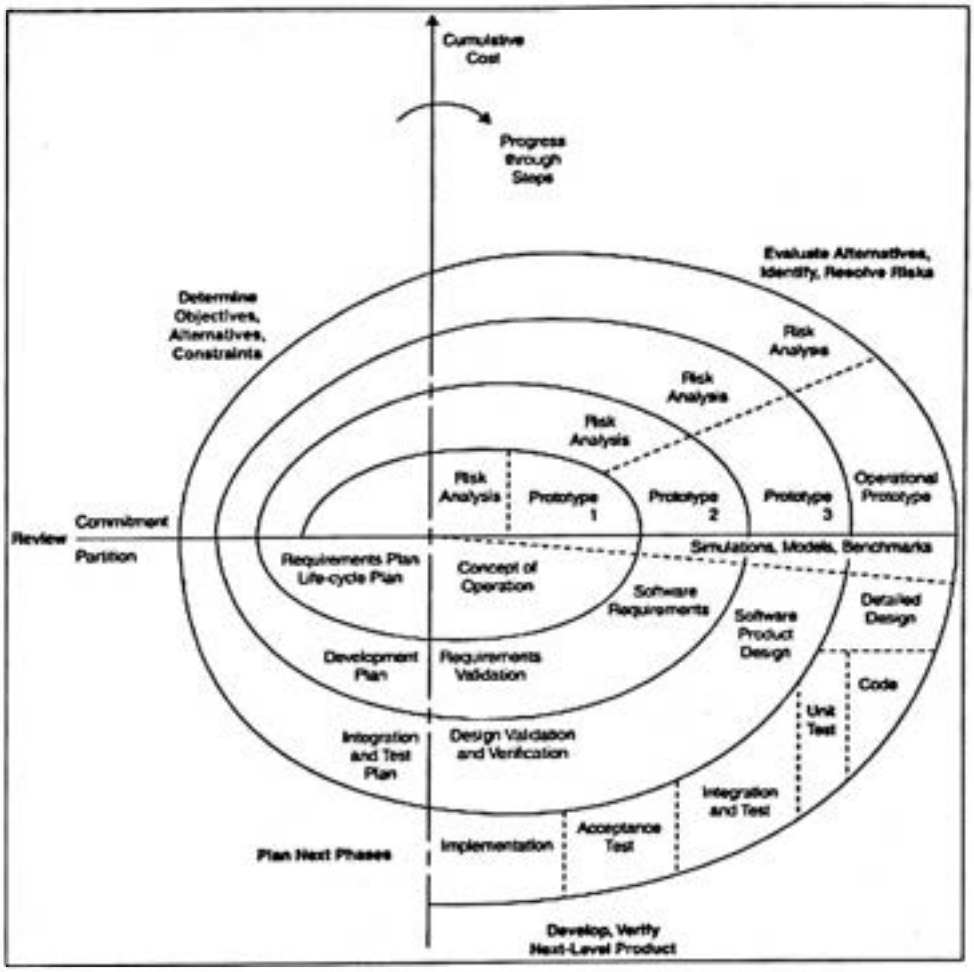

Figura 23 – O modelo espiral de desenvolvimento de *software*

FONTE: KAN, 2003, p.22

Boehm (1988) fornece uma discussão das vantagens e das desvantagens do modelo espiral. Suas vantagens são como segue: (KAN, 2003, p. 23)

- Seu leque de opções acomoda as características boas de modelos existentes de processos de *software*, visto que sua abordagem orientada a risco evita muitas de suas dificuldades. Boehm discute também as circunstâncias sob as quais este modelo se torna equivalente a outros modelos tais como o *waterfall* e o modelo evolucionário do protótipo (evolutionary prototype model);
- Foca a atenção inicial nas opções que envolvem reutilizar *software* existente. Estas opções são incentivadas porque a identificação e a avaliação antecipada das alternativas são etapas chaves em cada ciclo espiral. Este modelo acomoda a preparação para a evolução, o crescimento, e as mudanças do life-cycle do produto de *software;*

- Fornece um mecanismo para incorporar objetivos da qualidade do *software* no desenvolvimento de produto do *software;*
- Foca antecipadamente na eliminação de erros e alternativas não atraentes;
- Não envolve abordagens separadas para o desenvolvimento do *software* e a melhoria do *software;*
- Fornece uma estrutura viável para o desenvolvimento integrando o hardware w o *software*. A abordagem orientada a risco pode ser aplicada tanto ao hardware como ao *software*.

Por outro lado, as dificuldades com o modelo espiral incluem o seguinte: (KAN, 2003, p. 24)

- Os contratos de *software* se apóiam muito no controle, pontos de verificação, e produtos intermediários para os quais o modelo *waterfall* é bom. O modelo espiral tem um grande aopoio na flexibilidade e na liberdade, conseqüentemente, mais apropriado para o desenvolvimento interno de *software*. O desafio é como conseguir a flexibilidade e a liberdade prescritas pelo modelo espiral sem perder o controle exigido pelos contratos;
- Confiar na perícia da gerência de risco: A abordagem orientada a risco é a espinha dorsal do modelo. A especificação orientada a risco endereça os elementos de risco elevado com muitos detalhes e deixa os elementos de baixo risco para serem elaborados nos estágios posteriores. Entretanto, uma equipe sem experiência pode também produzir uma especificação oposta: grande nível de detalhes para os elementos bem-compreendidos, e de baixo risco, e pouca elaboração para os elementos mal compreendidos e de alto risco. Em tal caso, o projeto pode falhar e a falha pode ser descoberta somente depois que os recursos principais já tiverem sido investidos. Um outro ponto é que uma especificação orientada a risco é dependente de pessoas. No caso onde um projeto produzido por um perito deve ser executado por não peritos, o perito deve fornecer documentação adicional;
- O modelo espiral descreve um modelo processo flexível e dinâmico que possa ser usado com suas maiores vantagens por desenvolvedores experientes. Para recursos não experientes e especialmente para projetos em grande escala, entretanto, as etapas na espiral devem ser elaboradas e mais especificamente definidas de modo que a consistência, acompanhamento, e controle possam ser obtidos. Tais elaborações e controles são especialmente importantes na área de gerência e análise de risco.

2.4.3 O modelo de processo Iterativo

A abordagem de melhoria iterativa (iterative enhancement IE) (Basili e Turner, 1975), ou o processo iterativo de desenvolvimento (IDP), foi definido para começar com um subconjunto dos requisitos e desenvolver um subconjunto do produto que satisfaça às necessidades essenciais dos usuários, provê um veículo para a análise e o treinamento dos clientes, e fornecem uma experiência de aprendizagem para o desenvolvedor. Baseado na análise de cada produto intermediário, a modelagem (*design*) e as exigências são modificadas ao longo de uma série de iterações para fornecer um sistema aos usuários que atenda suas necessidades com modelo baseado feedback e aprendizagem. (KAN, 2003, p. 24)

O modelo IDP combina prototipação com a força do modelo clássico *waterfall*. Outros métodos tais como, análise de domínio e a análise de risco podem também ser incorporados no modelo IDP. O modelo tem muito em comum com o modelo espiral, especialmente no que diz respeito à prototipação e gerência de risco. Certamente, o modelo espiral pode ser considerado como um modelo específico de IDP. O modelo fornece também uma estrutura para muitos sistemas e métodos modernos de tecnologia de programação e técnicas tais como reutilizar, desenvolvimento orientado a objetos, e prototipação rápida. (KAN, 2003, p. 24-25)

A Figura 24 mostra um exemplo do modelo iterativo do processo do desenvolvimento usado por IBM Owego, New York. Com a finalidade de "construir um sistema evoluindo um protótipo arquitetural com uma série de versões executáveis, com cada experiência incorporando da iteração sucessiva e mais funcionalidades do sistema," a execução do exemplo contem oito etapas principais (Luckey *et al*, 1992): (KAN, 2003, p. 25)

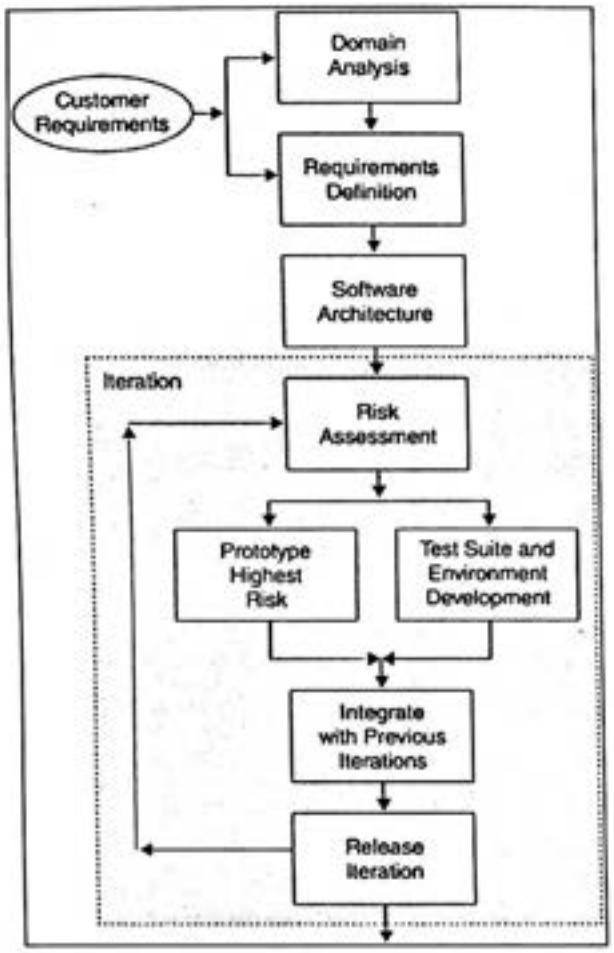

Figura 24 - Um exemplo do modelo de processo de desenvolvimento iterativo

FONTE: KAN, 2003, p. 25

a) análise de domínio (*domain analysis*);
b) definição de requisitos (*requirements definition*);
c) arquitetura do *software* (*software architecture*);
d) análise de risco (*risk analysis*);
e) protótipo (*prototype*);
f) suíte de Teste e ambiente de desenvolvimento (*test suite and environment development*);
g) integração com iterações prévias (*integration with previous iterations*);
h) liberação da iteração (*release of iteration*).

Como ilustrado na figura, o processo da iteração envolve as últimas cinco etapas; a análise do domínio, a definição dos requisitos, e a arquitetura do *software* são as etapas pré-iteração, que são similares àquelas no modelo *waterfall*. Durante as cinco etapas da iteração, as seguintes atividades ocorrem: (KAN, 2003, p. 26)

- Analise ou revisão dos requisitos do sistema;
- Projeto (desgin) ou recisão da solução que melhor satisfaz os requisitos;
- Identificação dos riscos mais elevados para o projeto e priorização dos mesmos; Mitigação dos riscos mais elevados através de prototipação, deixando uns riscos mais baixos para iterações subseqüentes;
- Definição e organização ou revisão das iterações seguintes;
- Desenvolvimento da suíte de teste da iteração e do ambiente de suporte ao teste;
- Execução da parcela do projeto que é requisito mínimo para satisfazer a iteração atuação;
- Integração do *software* no ambiente de testes e execução do teste de regressão;
- Atualização dos documentos para liberação junto com a iteração;
- Liberação da iteração.

O autor chama a atenção para o fato da extrema importância do desenvolvimento da suíte de testes ao longo da modelagem e desenvolvimento para a verificação da qualidade de cada iteração. No entanto, segundo o autor, na prática esta atividade nem sempre é enfatizada apropriadamente. (KAN, 2003, p. 26-27)

2.4.4 O Processo de desenvolvimento orientado a objetos

A abordagem orientada à objeto (OO) de projetar e programar, que foi introduzida nos anos 80, representa uma grande quebra de paradigma no desenvolvimento de *software*. Diferentemente da programação tradicional, que separa dados e controle, a programação orientada a objetos é baseada nos objetos, onde cada um deles é um conjunto de dados e operações (métodos) definidos que podem ser realizadas naqueles dados. Como o paradigma da modelagem estrutural e da decomposição funcional, a abordagem object-oriented transformou-se em um marco principal da tecnologia de programação. Nos primórdios da tecnologia de implantação OO (final dos anos 80 até meados dos anos 90), a maior parte da literatura de OO apresentou métodos de análise e modelagem; no entanto havia pouca informação sobre processos do desenvolvimento OO. Nos anos mais recentes a tecnologia orientado a objetos foi amplamente aceita e o desenvolvimento OO é agora tão difundido que não há mais questionamentos quanto a sua viabilidade. (KAN, 2003)

Branson e Herness (1992) propuseram um processo do desenvolvimento de OO para projetos em grande escala que se centrasse em uma metodologia de oito etapas suportada por um mecanismo de acompanhamento, em uma série das inspeções, em um conjunto de tecnologias, e em regras para prototipação e testes. O processo da oito-etapa é dividido em três fases lógicas: (KAN, 2003, p. 27-28)

a) A fase de análise focaliza em obter e em representar as exigências dos clientes em uma maneira concisa, visualizar um sistema essencial que represente as exigências de usuários não obstante a plataforma de implantação (ambiente de hardware e de *software*);
b) A fase de modelagem envolve modificar o sistema essencial de modo que possa ser implementado em um conjunto de *software* e hardware. As classes são combinadas e refinadas em uma hierarquia de desenvolvimento de classe. Os objetivos da síntese da classe são optimizar a reutilização e criar classes reutilizáveis;
c) A fase da implementação leva as classes definidas à sua conclusão.

As oito etapas do processo podem ser resumidas em: (KAN, 2003, p. 28)

a) Modelagem do sistema essencial;
b) Derivação das classes candidatas essenciais;
c) Confinamento do modelo essencial;
d) Derivação das classes adicionais;
e) Síntese das classes;
f) Definição das relações;
g) Término do projeto;
h) Implementação da solução.

A fase de análise do processo consiste em etapas 1 e 2, a fase do projeto consiste em etapas 3 a 6, e a fase da implantação consiste em etapas 7 e 8. Diversas iterações são esperadas durante a análise e o projeto.

Além da metodologia, requisitos, modelagem (*design*), análise, implementação, prototipação, e verificação, Branson e Herness (1993, apud Kan, 2003) afirmam que a arquitetura OO do processo de desenvolvimento deve também endereçar elementos como reutilização, ferramentas CASE, integração, construção e teste, e gerência de projeto. O modelo de processo de Branson e de Herness, baseado em sua experiência object-oriented em IBM Rochester, representa uma tentativa de implantar a tecnologia object-oriented em organizações grandes. (KAN, 2003, p. 29)

Finalmente, o elemento reutilização merece mais discussão na perspectiva do processo. Reutilização do modelo (*design*) e do código dá ao desenvolvimento object-oriented vantagens significativas na qualidade e na produtividade. Entretanto, reutilização não é obtido automaticamente simplesmente usando desenvolvimento orientado à objeto. O desenvolvimento orientado à objeto fornece uma fonte potencial grande dos componentes reusáveis, que devem ser generalizados para se tornar aproveitáveis em ambientes novos do desenvolvimento. Em termos do ciclo de vida do desenvolvimento, a generalização para reutilizar é considerada tipicamente um "add-on" no fim do projeto. Entretanto, as atividades da generalização consomem tempo e recursos. Conseqüentemente, desenvolver com reutilização é o que cada projeto OO busca, mas desenvolver para reutilizar é difícil de executar. Este paradoxo da reusabilidade, segundo o autor, explica o fato de não existir uma quantidade significativa de código reusável no nível do negócio, apesar das promessas da tecnologia OO, embora haja muitas bibliotecas reutilizáveis para propósito geral.

Conseqüentemente, as organizações que pretendem aproveitar a vantagem de reutilização do desenvolvimento OO devem cuidar deste assunto em seu processo do desenvolvimento. (KAN, 2003, p. 29-30)

2.4.5 O processo unificado de desenvolvimento da Rational Software (RUP)

Em 1997, o processo unificado de desenvolvimento de *software*, que foi desenvolvido por Jacobson, por Booch, e por Rumbaugh (1997), propriedade da Racional *Software* Corporation, hoje, uma linha de negócios da IBM, foi publicado. O processo apoia-se no UML como padrão de linguagem de modelagem visual. Trata-se de um processo orientado à casos de uso, centrado na arquitetura (*architecture-centric*), iterativo, e incremental. Os casos do uso são os componentes chave que dirigem este modelo de processo. Um caso de uso é uma parte ou conjunto de funcionalidades que dão ao usuário um resultado de valor. Todos os casos de uso desenvolvidos podem ser combinados em um modelo de caso de uso, que descreve o funcionamento completo do sistema. O modelo de caso de uso é análogo à especificação funcional no processo tradicional de desenvolvimento de *software*. Os casos de uso são desenvolvidos junto aos usuários e modelados em UML. Estes representam os requisitos para o *software* e são utilizados ao longo do processo de desenvolvimento. O processo unificado é também descrito como centrado na arquitetura (*architecture-centric*). Esta arquitetura é uma visão de todo o modelo (*design*) com as caracteristicas importantes tornadas evidentes "retirando-se" os detalhes. Ela trabalha em conjunto com os casos de uso. Subsistemas, classes e componentes são expressos na arquitetura e são também modelados em UML. Por fim, o Processo Unificado é iterativo e incremental. As iterações representam passos em um fluxo de trabalho, e os incrementos geram o crescimento nas funcionalidades do produto (KAN, 2003, p.30-32).

Os principais fluxos de trabalho (workflows) para o desenvolvimento iterativo são listados a seguir, bem como mostrados na Figura 25: (KAN, 2003, p. 32)(RATIONAL, 2002)

- Requisitos
- Análise e modelagem (*Design*)
- Implementação
- Teste

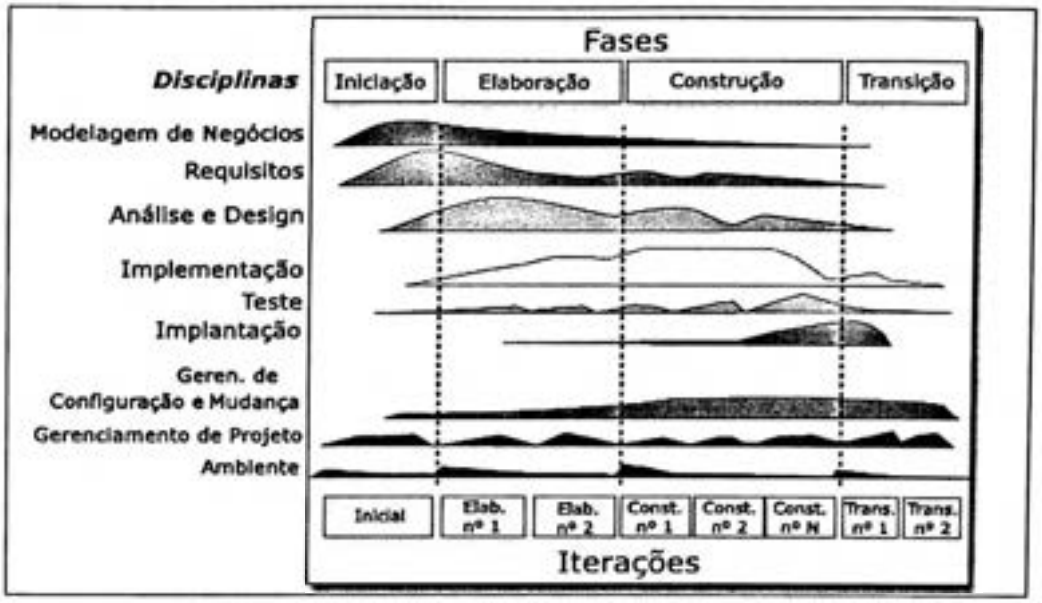

Figura 25 - Fluxos de trabalho do processo unificado de desenvolvimento

FONTE: RATIONAL, 2002

O processo unificado consiste em ciclos. Cada ciclo resulta na liberação de um novo *release* do sistema, e cada *release* é um produto entregável. Cada ciclo de desenvolvimento tem quatro fases: iniciação, elaboração, construção, e transição. Um número de iterações ocorre em cada fase, e os fluxos de trabalho principais ocorrem em cada uma das quatro fases. (KAN, 2003, p. 30)

Durante a iniciação, uma boa idéia para o produto *software* é desenvolvida e o projeto é iniciado. Um modelo de caso de uso simplificado é criado e os riscos do projeto são priorizados. A seguir, durante a fase de elaboração, os casos de uso são especificados em detalhes e a arquitetura do sistema é desenhada. O gerente de projeto começa a planejar e a estimar as atividadades para os recursos. Todas as visões do sistema são entregues, incluindo o modelo de casos de uso, o modelo do produto, e o modelo de implementação. Estes modelos são desenvolvidos em UML e mantidos em um gerenciamento de configuração. Uma vez que esta fase está completa, a fase da construção começa. A partir da arquitetura, o projeto cresce em direção ao sistema completo. O código é desenvolvido e o *software* é testado. Então o *software* é avaliado para determinar se o produto atende as necessidades dos usuários de modo que alguns clientes possam obter uma versão antecipada do *software*. Finalmente, a fase

da transição começa com o beta teste. Nesta fase, os defeitos são descobertos, acompanhados e reparados e o *software* passa para uma equipe da manutenção. (KAN, 2003, p. 30)

2.4.6 O processo XP

Um processo muito controverso de OO que ganhou o reconhecimento e gerou debates vigorosos entre Engenheiros de *Software* é o *Extreme Programming* (XP), proposto por Kent Beck (2000) (apud Kan, 2003, p. 31-32). Este processo leve, iterativo e incremental tem quatro marcos de valores principais: comunicação, simplicidade, realimentação (*feedback*) e coragem. Com esta fundação, o XP prega as seguintes práticas: (KAN, 2003, p. 31-32)

- O jogo do planejamento: os times de desenvolvimento estimam o tempo, risco, e a ordem da "história". O cliente define o escopo, as datas de liberação e a prioridade;
- Metáfora do sistema: uma metáfora descreve como o sistema funciona;
- Modelagem simples: os modelos são mínimos, apenas o suficiente para passar nos testes que limitam o escopo;
- Programação em pares: toda a modelagem e programação é realizada por duas pessoas em uma estação de trabalho. Isto permite difundir melhor os conhecimentos e utilizar constantemente as revisões em pares;
- Testes unitários e de aceite: os testes unitários são escritos antes da codificação para dar um entendimento claro do código e prover uma biblioteca completa de testes;
- Re-manufatura: o código é re-manufaturado antes de ser implementada uma funcionalidade, de forma a ajudar a manter o código "limpo" e organizado;
- Propriedade coletiva do código: através da troca de times para a visualização de todas as partes do código, todos os desenvolvedores estão aptos a reparar partes com defeitos;
- Integração contínua: quanto mais um código é integrado, mais provavelmente ele se manterá funcionando sem grandes paradas;
- Cliente *on-site*: um cliente *on-site* é considerado parte do time e é responsável pelo conhecimento do domínio e pelo teste de aceite;
- Semana de 40 horas: estipular uma semana de 40 horas garante que os desenvolvedores estarão sempre alertas;
- Pequenas liberações: as versões liberadas são pequenas, no entanto contém funcionalidades úteis;

- Padronização da codificação: os padrões de codificação são definidos pelo time que mantém a aderência a ele.

De acordo com o Beck, pelo fatos destas práticas se balancearem e se reforçarem mutuamente, implementar todas elas juntamente é o que faz o XP extremo. Com estas práticas, o time de engenheiros de *software* podem "abraçar as mudanças". Ao contrário de outros modelos de proessos evolucionários, o XP desencoraja a obtenção preliminar de requisitos, a análise extensiva e a modelagem do produto. Ao invés disto, ele limita o planejamento intencionalmente para futura flexibilidade, promovendo a filosofia YANGI (*You Aren't Gonna Need It* – Você não vai precisar dele) que enfatiza a utilização de menos classes e documentação reduzida.

Kan (2003, p. 32) descreve que, em sua percepção, a filosofia e as práticas de XP parecem ser mais aplicáveis aos projetos pequenos. Para desenvolvimentos grandes e complexos de *software*, alguns princípios do XP tornam-se mais difíceis de implementar e podem até mesmo correr contra alguns desejos tradicionais no qual se apóiam os projetos bem sucedidos. Beck menciona que os esforços em XP têm obtidos melhores resultados com times de 10 ou menos membros. (KAN, 2003, p. 32)

2.4.7 Benefícios da melhoria de processos de desenvolvimento de *software*

Estouro de prazos, custos e baixa qualidade têm sido endêmico na indústria de *software* por mais de 50 anos. (KAN, 2003, p. 453)

Um estudo profundo realizado em 13 organizações pelo SEI (*Software Engineering Institute*) encontrou que uma organização típica (média) obteve, com a melhoria sistemática dos processos de desenvolvimento de software, na média, um ganho de produtividade de 35% ao ano, uma redução de cronograma de 19% ao ano e uma redução de defeitos pós lançamento de 39% ao ano. Estes resultados estão resumidos na tabela 13.1 (MCCONNELL, 2003, p. 131).

Considerando-se os ganhos obtidos pelas melhores organizações, os resultados são ainda superiores. A organização com o maior ganho de produtividade melhorou 58% ao ano em um

período de 4 anos, para um ganho combinado de mais de 500%. A melhor redução de cronograma foi de 91%. A maior melhoria de qualidade de longo-prazo foi a redução da taxa de defeitos em produção (pós-lançamento) em 39% ao ano por 9 anos, em um total de redução de defeitos combinado de 99%. Duas organizações atingiram reduções de defeito no curto prazo de 70% ou mais em menos de 2 anos (MCCONNELL, 2003, p. 131).

Tabela 10 - Resultados de esforços para melhoria de processos

Fator	Melhoria média	Melhores resultados
Produtividade	35%/ano	58%/ano
Cronograma	19%/ano	23%/ano
Defeitos reportados após liberação	39%/ano	39%/ano[12]
Valor para o negócio da melhoria organizacional (*Business value of organizational improvement*)	500 %	880 %

FONTE: MCCONNELL, 2003, p. 131

As organizações que se apoiam em processos de desenvolvimento "codificar-e-reparar" (*code-and-fix*) tendem a pensar que existe um *tradeoff* entre baixa contagem de defeitos e produtividade. No entanto, o autor afirma que a maior parte dos custos de um projeto *code-and-fix* são decorrentes de um trabalho de correção não planejado. Os resultados na Tabela 13 confirmam que para a maior parte das organizações não há *tradeoff* entre maior produtividade e melhor qualidade. Organizações que focam na prevenção de defeitos também obtêm cronogramas mais enxutos e maior produtividade (MCCONNELL, 2003, p. 132).

Em termos percentuais, o número de empresas que se engajaram na melhoria sistemática de suas práticas de desenvolvimento de *software* é pequeno. A Tabela 11 sumariza o retorno sobre o investimento reportado por 20 organizações (MCCONNELL, 2003, p. 132).

Tabela 11 - Resultados de esforços para a melhoria de processos

Organização	Resultados
BDN International	ROI 300%
Boeing Information Systems	US$ 5,5 milhões economizados em 1 ano, ROI 775%
Computer Sciences Corporation	65% de redução nas taxas de defeitos
General Dynamics Decision Systems	70% de redução no retrabalho; 94% de redução na taxa de defeitos; ganho de produtividade de 2.9 x
Harris ISD DPL	90% de redução na taxa de defeito; ganho de produtividade de 2.5 x
Hewlett-Packard SESD	ROI 900%
Hughes	US$2 milhões de redução annual nos custos, ROI 500%
IBM Toronto	90% de redução nos defeitos entregues, 80% de redução no retrabalho
Motorola GED	2–3 x de melhoria na produtividade, 2–7 x redução do tempo dos ciclos,

	ROI 677%
Philips	ROI 750%
Raytheon	ROI 770%
Schlumberger	Redução de 4 x nos *bugs* dos testes beta
Siemens	90% de redução nos defeitos liberados
Telcordia	Redução na taxa de defeitos para 1/10 em relação à indústria; aumento na satisfação do consumidor de 60% para 91% em 4 anos
Texas Instruments—Systems Group	90% de redução nos defeitos entregues
Thomson CSF	ROI 360%
U.S. Navy	ROI 410%
USAF Ogden Air Logistics Center	ROI 1,900%
USAF Oklahoma City Air Logistics Center	ROI 635%
USAF Tinker Air Force Base	ROI 600%

FONTE: MCCONNELL, 2003, p. 133

McConnell (2003, p. 133) diz que os detalhes de como as organizações obtiveram maior eficiência em desenvolvimento de *software* variam entre diferentes organizações. Não obstante, algumas práticas específicas foram encontradas como sendo geralmente eficazes. A Tabela 12 mostra alguns retornos para algumas práticas escolhidas.

Tabela 12 - Retorno sobre o investimento para práticas de *software* escolhidas

Prática	ROI em 12 meses	ROI em 36 meses
Inspeções formais de código	250%	1200%
Inspeções formais de modelo	350%	1000%
Planejamento de tecnologia de longo prazo	100%	1000%
Ferramentas de estimativa de custo e qualidade	250%	1200%
Medidas de produtividade	150%	600%
Avaliações de processos	150%	600%
Treinamento gerencial	120%	550%
Treinamento técnico	90%	550%

FONTE: MCCONNELL, 2003, p. 134

Já Kan (2003) examinou mais de 10000 projetos de *software* desde 1984 juntamente com o *Software Productivity Research* (SPR) espalhados em todos os 5 níveis do CMM (*SEI capability maturity model*), em sua maior parte advindos de 600 empresas e agências de governo. Os resultados foram publicados em diversos livros que endereçam os aspectos de economia e de qualidade da melhoria do processo de desenvolvimento de *software* (Jones 1994, 1996, 1997, 1998, e 2000 apud Kan, 2003, p. 454).

Kan (2003) descreve uma classificação em estágios através das quais a melhoria de processo ocorre de acordo com sua observação, nas empresas examinadas que obtiveram os melhores resultados. (KAN, 2003, p. 454)

a) Estágio 0: avaliação do processo de desenvolvimento de *software*, estabelecimento de uma base e realização de *benchmark*;
b) Estágio 1: foco em tecnologias gerenciais;
c) Estágio 2: foco em metodologias e processos de desenvolvimento de *software*;
d) Estágio 3: foco em novas ferramentas e abordagens;
e) Estágio 4: foco em infra-estrutura e especialização;
f) Estágio 5: foco em reusabilidade;
g) Estágio 6: foco em liderança na indústria.

Kan (2003, p. 454) ressalta que a avaliação (*assessment*) por si só não gera nenhuma melhoria, de forma que ela se encontra fora dos 6 estágios numerados de melhoria. Segundo ele, uma avaliação de processo de *software* é análoga ao estudo de um diagnóstico médico, já que não cura nenhuma doença, mas alimenta os médicos com informações necessárias para planejar as terapias eficientes.

Estes seis estágios provêm uma estutura para estratégias de melhoria de processo de desenvolvimento de *software*. No entanto, as empresas são diferentes e portanto as especificidades da estratégia de melhoria de cada uma devem ser aderentes à sua cultura local e às suas necessidades particulares (Grady, 1997 apud Kan, 2003, p. 454).

A Tabela 13 apresenta os custos de melhoria de processos para cada um destes estágios de acordo com os estudos realizados. Estas informações são baseadas no tamanho das empresas em termos de pessoas ligadas ao desenvolvimento de *software*. Os dados de custo estão expressos em termos de "custo por pessoa" ou os custos aproximados para cada funcionário nos departamentos de *software*. Os elementos de custo incluem treinamento, taxas de consultoria, equipamentos, licenças de *software*, e melhorias nas condições do escritório (KAN, 2003, p. 459-460).

Tabela 13- Despesas com a melhoria de processo *per capita* em US$

Estágio	Descrição	Time pequeno (< 100)	Time médio (< 1000)	Time grande (< 10000)	Time gigante (> 10000)	Média
0	Avaliação	100	125	150	250	156
1	Gerenciamento	1500	2500	3500	5000	3125
2	Processo	1500	2500	3000	4500	2875
3	Ferramentas	3000	6000	5000	10000	6000
4	Infra-estrutura	1000	1500	3000	6500	3000
5	Reutilização	500	2500	4500	6000	3375
6	Liderança na Industria	1500	2000	3000	4500	2750
Total de despesas		9100	17125	22150	36750	21281

FONTE: KAN, 2003, p. 459

Os tamanhos na Tabela 13 referem-se às populações de *software*, e dividem as organizações em 4 domínios de tamanho: menores que 100 pessoas ligadas a *software*, menores que 1000, menores que 10000, e maiores que 10000, as quais são organizações gigantes de *software* tais como IBM, Accenture Consulting, e Electronic Data Systems (EDS), que possuem mais de 50000 pessoas ligadas a *software* na corporação. (KAN, 2003, p. 460).

A tabela mostra que as avaliações de processo de *software* são relativamente pouco onerosas, no entanto a melhoria dos processos e as ferramentas necessárias após as avaliações podem ser bem caras.

Um outro ponto importante é o tempo que leva para se mover em cada estágio na seqüência de melhoria de processo. A Tabela 14 ilustra o número aproximado de meses dedicados em mover-se de estágio para estágio. Empresas menores podem se mover muito mais rapidamente do que empresas grandes e agências governamentais. Empresas grandes freqüentemente possuem muita burocracia e muitos níveis de aprovação. (KAN, 2003, p. 460)

Kan (2003, p. 460) menciona que em situações com existência de polarização de opinião ou oposição política, o progresso pode ser muito lento ou até mesmo não existir.

Tabela 14 - Estágios da melhoria de processo em meses

Estágio	Descrição	Time pequeno (< 100)	Time médio (< 1000)	Time grande (< 10000)	Time gigante (> 10000)	Média
0	Avaliação	2.00	2.00	3.00	4.00	2.75
1	Gerenciamento	3.00	6.00	9.00	12.00	7.50
2	Processo	4.00	6.00	9.00	15.00	8.50

3	Ferramentas	4.00	6.00	9.00	12.00	7.75
4	Infra-estrutura	3.00	4.00	9.00	12.00	7.00
5	Reutilização	4.00	6.00	12.00	16.00	9.50
6	Liderança na Industria	6.00	8.00	9.00	12.00	8.75
Soma (pior caso)		26.00	38.00	60.00	83.00	51.75
Sobreposição (melhor caso)		16.90	26.60	43.20	61.42	33.64

FONTE: KAN, 2003, p. 460

Na opinião de Kan (2003, p. 461), uma questão importante é qual o tipo de valor ou o retorno sobre investimento que irá ocorrer na melhoria de processo de desenvolvimento de *software*. A Tabela 15 mostra somente melhorias aproximadas de prazos, custos e qualidade (esta definida, neste contexto, como níveis de defeito no *software*). Os resultados são expressos em percentual de melhorias comparado com o patamar inicial no começo do processo de melhoria (KAN, 2003, p. 461).

> Os melhores projetos nas melhores empresas são capazes de implantar *software* com somente 5% de defeitos latentes em relação a projetos similares em outras. As taxas de produtividade são superiores a 300% e os prazos possuem em torno de um quarto de extensão. Estas diferenças notáveis podem ser utilizadas para justificar investimentos em atividades de melhoria de processos de desenvolvimento. (KAN, 2003, p. 461)

Os maiores benefícios não ocorrem até o estágio 5, conforme pode-se observar, quando um programa completo de reusabilidade é implementado. Pelo fato da reusabilidade ter o melhor retorno e os maiores resultados na visão de Kan (2003, p. 461), freqüentemente se questiona o porquê de não implantá-la no primeiro estágio. A justificativa, de acordo com Kan (2003, p. 461), da implantação da reusabilidade apenas no estágio 5 é que um programa de reusabilidade bem sucedido depende do domínio da qualidade de *software*. O controle efetivo da qualidade de *software* implica em utilizar uma série de processos e tecnologias precursoras, como inspeções formais, planos de testes formais, grupos formais de garantia da qualidade e processos formais de desenvolvimento. Enquanto a qualidade de *software* não estiver no nível de estado da arte, qualquer tentativa de reutilização pode ser perigosa, pois a reutilização de partes que contenham sérios erros irão resultar em prazos e custos maiores em relação a não reutilização destes artefatos.

Tabela 15 - Melhorias nos níveis de defeito no *software*, produtividade e cronograma

Estágio	Descrição	Defeitos entregues (%)	Produtividade do desenvolvimento (%)	Cronograma de desenvolvimento (%)
0	Avaliação	0.00	0.00	0.00
1	Gerenciamento	-10.00	10.00	-12.00
2	Processo	-50.00	30.00	-17.00
3	Ferramentas	-10.00	25.00	-12.00
4	Infra-estrutura	-5.00	10.00	-5.00
5	Reutilização	-85.00	70.00	-50.00
6	Liderança na Industria	-5.00	50.00	-5.00
	Total	-95.00	365.00	-75.00

FONTE: KAN, 2003, p. 461

Para ilustrar medidas de melhoria de processo no nível de atividades, Kan (2003, p. 462-467) apresenta um estudo comparativo realizado entre duas organizações com níveis diferente de maturidade de processos (CMM nível 1 e nível 3). A tabela 9 ((T)) mostra, lado a lado, uma comparação dos custos por atividade no desenvolvimento de *software* entre uma organização CMM nível 1 e nível 3, a partir de um estudo apresentado por Kan (2003, p. 462-467), levando-se em conta o desenvolvimento de um *software* equivalente por ambas, ou seja, utilizando-se a mesma linguagem de programação (C), o mesmo número de pontos de função (1000), o mesmo número de linhas de código (125000), o mesmo número de horas trabalhadas por mês (132) e a mesma média de salário mensal (US$ 7500). A tabela mostra uma redução geral de custo em torno de 20% entre o nível 1 e o nível 3. No entanto, a atividade responsável pela maior economia é o Teste (KAN, 2003, p. 462-467).

A comparação apresentou custos das inspeções mais elevados, ao invés de menores, para o nível 3 do CMM. Alguns custos, tais como aqueles ligados à documentação do usuário, são os mesmos em ambos os cenários. Observando-se a qualidade, os resultados mostram que a melhoria tanto na prevenção de defeitos como na remoção destes no nível 3 causam uma redução significativa nos defeitos entregues. O número reduzido de defeitos possibilita ciclos de desenvolvimento menores e mais eficientes em termos de custos. Quando os projetos correm com atrasos, freqüentemente os problemas passam desapercebidos até o início dos testes. Quando grandes defeitos são encontrados durante o teste, é muito tarde para trazer o projeto de volta ao controle. O objetivo é previnir os defeitos e eliminá-los antes dos testes. (KAN, 2003, p. 463-464)

Tabela 16 - Comparação dos custos das atividades (em US$)

Atividade	SEI CMM nível 1	SEI CMM nível 3	Variação nos custos	Variação Percentual
Requisitos	68.182	60.000	-8.182	-12.00
Modelagem	166.667	150.000	-16.667	-10.00
Revisões de modelagem	21.429	50.000	28.571	133.33
Código	441.176	340.909	-100.267	-22.73
Inpeções de código	30.000	60.000	30.000	100.00
Teste	468.750	187.500	-281.250	-60.00
Garantia da qualidade	75.000	125.000	50.000	66.67
Documentação	62.500	62.500	0	0.00
Gerenciamento	202.703	187.500	-15.203	-7.50
Total	1.536.406	1.223.409	-312.997	-20.37%
Custo por ponto de função (FP)	1.536.41	1.223.41	-313.00	-20.37%
Custo por LOC	12.29	9.79	-2.50	-20.37%

Classe de aplicação	Sistema de *software*
Linguagem de programação	C
Tamanho em pontos de função (FP) [13]	1.000
Tamanho em linhas de código (LOC) [14]	125.000
Horas de trabalho por mês	132
Média mensal de salário	US$ 7.500

FONTE: KAN, 2003, p. 465

A Tabela 17 ilustra as diferenças em potenciais de defeito, níveis de eficiência na remoção de defeitos e defeitos entregues nos dois casos. Conforme pode-se notar, uma combinação de prevenção de defeitos e remoção dos mesmos pode render reduções significativas nos níveis de defeitos entregues. Pelo fato de encontrar e fixar os defeitos serem as atividades mais custosas e consumidoras de tempo para o desenvolvimento de *software*, os projetos que são bem sucedidos na prevenção de defeitos ou remoção dos mesmos via inspeções obterão cronogramas menores e maiores produtividades, bem como melhor qualidade. (KAN, 2003, p. 465)

A análise, baseada no custo das atividades, ilustra que a melhoria de processos não cria uma melhoria homogênea para cada atividade. As melhorias tendem a ser mais significativas para atividades chave como teste mas pouco visíveis para outras atividades como documentação do usuário. (KAN, 2003, p. 465-466)

---

[13] FP = *function points per IFPUG Counting Practices Manual 4.1.*
[14] LOC = *lines of noncommentary code.*

Tabela 17 - Diferenças de defeito entre o CMM nível 1 e 3

Nível	Defeitos potenciais[15]	Eficiência de remoção [16](%)	Defeitos entregues	Defeitos por ponto de função[17]	Defeitos por KLOC[18]
SEI nível 1	6150	85.01%	922	0.92	7.38
SEI nível 3	3500	95.34	163	0.16	1.30

FONTE: KAN, 2003, p. 466

Certamente, para um número de atividades importantes tais como modelagem, inspeções de código e trabalhos de garantia de qualidade, os custos serão maiores para as organizações mais maduras (CMM nível 3) do que aquelas no nível 1. (KAN, 2003, p. 466)

2.5 Revisão

As revisões são a primeira e mais primária forma de atividade de controle da qualidade, e, são mais eficientes em termos de custo, já que consomem menos recursos em relação aos testes – possuem uma duração mais curta, requerem menos recursos humanos, podem ser agendadas e reagendadas com impactos mínimos. As revisões são conduzidas durante o desenvolvimento e não ao final, como no caso dos testes, para verificar se os produtos de cada fase estão corretos em relação às suas entradas e atividades. Às vezes as revisões são conhecidas como testes pré-codificação, no entanto os testes necessitam da execução do código enquanto as revisões não (HORCH, 2003, p. 53).

As revisões pode ser realizadas de diversas maneiras, como revisões aos pares informais (*informal peer reviews*), tal como um-a-um (*one-on-one*), *walk-throughs*, inspeções (*inspections*) ou auditorias (*audits*). Podem também ser revisões formais como revisões de verificação (*verification reviews*), revisões de ponto de decisão (*decision-point reviews*), auditorias físicas ou funcionais (*physical* ou *functional audits*), ou revisões pós-implementação (*post-implementation review*) (HORCH, 2003, p. 53).

[15] Defeitos potencialmente encontrados desde os requisitos até pelo menos um ano de uso do cliente.
[16] Percentual dos defeitos potenciais encontrados antes da entrega do *software* ao cliente.
[17] *Function points per IFPUG Counting Practices Manual 4.1.*
[18] KLOC = 1000 linhas de código.

Boehm apud (HORCH, 2003, p. 53) e outros desenvolveram um gráfico similar ao da Figura 26 que mostra o custo crescente dos defeitos a medida que ele permanece no produto ao longo de seu ciclo de vida. A idéia da revisão é encontrar os defeitos a medida que eles vão sendo criados, ao invés de depender do teste ou uso do software para descobri-los. Isto possibilita um impacto menor de custo no projeto ocasionado pelos defeitos (HORCH, 2003, p. 53).

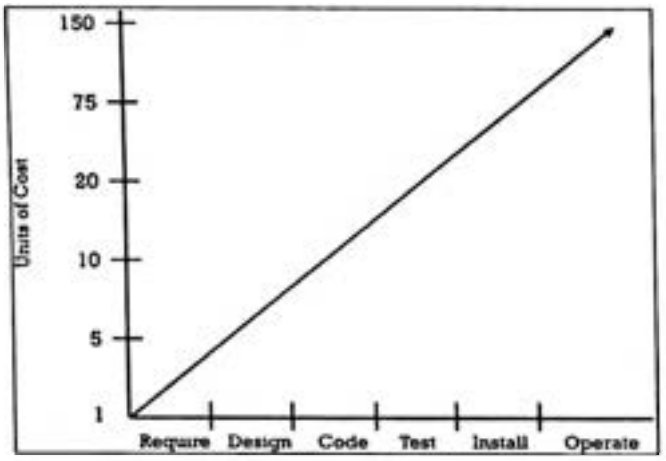

Figura 26 - Custo da identificação de defeitos
FONTE: HORCH, 2003, p. 53

Cada revisão tem uma finalidade, um objetivo, uma audiência, e um tipo de participante. Algumas revisões podem suportar vários times durante o desenvolvimento, tal como os *walk-throughs* de modelagem. Outras, como as auditorias funcionais, são de tal magnitude que normalmente são eventos de uma vez apenas que formam a base para decisões maiores sobre o produto (HORCH, 2003, p. 53).

As revisões são classificadas em (HORCH, 2003, p. 54):

a) Revisões *in-process* (geralmente consideradas informais);
b) Revisões de fim de fase (*phase-end*) (geralmente consideradas formais).

As revisões *in-process* são revisões informais conduzidas durante cada fase do SDLC (*Software* Development Life Cycle). Informais pelo fato de existir pouca informação cliente sobre os resultados da atividade. São conhecidas como revisões em pares por serem normalmente conduzidas por pares de desenvolvedores (HORCH, 2003, p. 54).

Uma regra para o agendamento é não revisar mais do que o desejo do desenvolvedor de "jogar tudo para longe" (HORCH, 2003, p. 54).

Outra regra é executar uma revisão *in-process* a cada duas semanas. A Figura 27 mostra sugestões da aplicação destas regras. Cada projeto irá determinar suas próprias regras apropriadas para a revisão in-process (HORCH, 2003, p. 54).

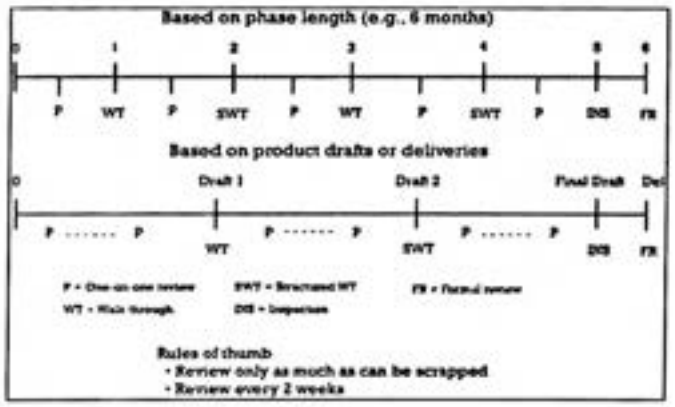

Figura 27 – Regras de agendamento

FONTE: HORCH, 2003, p. 54

Existe uma escala de rigor nas revisões *in-process*, onde a menos rigorosa é a revisão em pares (*peer review*), seguida pelo *walk-through*, enquanto a mais rigorosa é a inspeção. A Tabela 18 mostra resumidamente um quadro comparativo entre algumas características das revisões *in-process*. (HORCH, 2003, p. 54).

Tabela 18- Características das revisões *in-process*

Tipo de revisão	Registros	Nível de CM	Participantes	Nível de Stress
Um a um (*One-on-one*)	Nenhum	Nenhum	Colega de trabalho	Muito baixo
Walk-through	Cópia com anotações ou relatórios de defeitos	Provavelmente nenhum	Membros do projeto com interesse	De baixo para médio
walk-through estruturado	Relatórios de defeitos	Informal	Membros do projeto selecionados	Médio
Inspeção	Banco de dados de relatórios de defeitos	Formal	Pessoas com papéis específicos	Alto

FONTE: HORCH, 2003, p. 54

2.5.1 Revisões um-a-um

A revisão um-a-um é a menos rigorosa e, por isto, geralmente a menos estressante. Nela, o desenvolvedor pede ao seu colega para checar o produto para ter certeza que ele está correto. O resultado da revisão é geralmente verbal, ou no máximo, algumas marcas são feitas no rascunho do produto (HORCH, 2003, p. 54).

Elas são geralmente mais utilizadas durante o esboço do produto, ou na correção de um defeito, e cobrem pequenas partes de cada vez. Pelo fato de não haver teoricamente nenhuma distribuição de registros dos defeitos e correções sugeridas, o desenvolvedor sente pouca pressão ou ameaça (HORCH, 2003, p. 54).

2.5.2 *Walk-throughs*

A medida que o desenvolvedor atinge certos pontos em seu trabalho, um grupo de pares pode ser requisitado para revisar o trabalho enquanto o desenvolvedor o descreve, ou navega através das funcionalidades do produto. Desta forma, defeitos podem ser encontrados e corrigidos imediatamente, antes do produto ser utilizado como base para as atividades da próxima fase. Pelo fato de ser normalmente conduzido de maneira informal e pelos pares do desenvolvedor, existe uma menor tendência por parte do produtor de ser defensivo e protecionista, possibilitando uma troca de informações mais aberta e, conseqüentemente, melhores resultados (HORCH, 2003, p. 54).

Os revisores são encorajados a comentar a abordagem do desenvolvedor e os resultados com a intenção de expor os defeitos no produto. Por outro lado apenas os desvios dos requisitos do produto estão abertos a crítica, de forma que a menos que o produto viole um requerimento, padrão, restrição ou produza um resultado errado, melhorias não são sugeridas nas reuniões *walk-through* (HORCH, 2003, p. 54).

Os resultados destas reuniões devem ser gravados em um relatório de defeitos de *software*, pois torna os defeitos encontrados um pouco mais público. Isto pode aumentar o estresse e ameaça ao desenvolvedor, por outro lado ajuda a garantir que todos os defeitos encontrados no *walk-through* sejam endereçados e corrigidos (HORCH, 2003, p. 54).

2.5.3 Inspeções

Uma outra revisão *in-process* mais rigorosa é a inspeção, que apesar de muito similar ao *walk-throughs*, requer um conjunto de participantes mais específicos e uma divulgação mais elaborada dos itens de ação. Diferentemente do walk-through, a inspeção requer um relatório escrito de seus resultados e uma estrita gravação dos relatórios de problemas (HORCH, 2003, p. 55).

Tipicamente uma inspeção tem dois encontros. O primeiro é um mini *walkthrough* do material por parte do produtor, com a intenção de passar um *overview* do produto e seu lugar no sistema. Então inspeções individuais são agendadas para revisar o material antes da inspeção propriamente dita. Na inspeção, o leitor guia os inspetores através do material e conduz a procura por defeitos. Na conclusão da inspeção, o relator prepara os relatórios requeridos e o moderador supervisiona o acompanhamento nas correções e modificações necessárias encontradas pelos inspetores (HORCH, 2003, p. 55).

Pelo fato de ser mais rigorosa, a inspeção tende a ser mais cara em termos de tempo e recursos do que o *walk-through* e é geralmente utilizada em projetos com maiores riscos e complexidade. No entanto, a inspeção é normalmente mais bem sucedida em encontrar defeitos do que o *walk-through*, e por isto algumas empresas somente utilizam este processo para suas revisões *in-process* (HORCH, 2003, p. 55).

Por fim, dado o seu rigor em armazenar os defeitos, ela é a mais estressante e ameaçadora dos processos de revisão *in-process* (HORCH, 2003, p. 55).

2.5.4 Auditorias *in-process*

As auditorias também podem ser informais enquanto o SDLC evolui. Elas são utilizadas para avaliar a aderência do produto aos padrões impostos por algumas metodologias de desenvolvimento de *software* quanto ao conteúdo da documentação e requisitos de formatação (HORCH, 2003, p. 56).

As revisões de fim de fase (*phase-end*) são formais e geralmente ocorrem ao final de uma fase do SDLC, ou período de trabalho, e estabelecem baselines para o trabalho nas fases subseqüentes. Diferentemente das revisões *in-process* que lidam com um simples produto, as revisões de fim de fase incluem o exame de todos os produtos do trabalho que foram realizados durante a fase em questão. Por exemplo a revisão dos requisitos de *software* (*software requirements review* - SRR) é um exame formal do documento de requisitos e estabelece as basesw para as atividades na fase seguinte de modelagem (HORCH, 2003, p. 56).

As revisões de fim de fase, na visão de Horch (2003, p. 56), deviam ser consideradas pontos de decisão para prosseguir ou não (*go-or-no-go*). Na revisão ao final de cada fase, deve existir informações de *status* suficientes para examinar o risco do projeto, incluindo sua viabilidade, a probabilidade que ele seja finalizado com o orçamento e prazo estimado, e a probabilidade dele atender seus objetivos originais (HORCH, 2003, p. 56).

A Figura 28 mostra várias revisões de fim de fase ao longo do SDLC. As revisões formais, como a revisão dos requisitos do *software*, revisão preliminar de modelagem (*design*) (*preliminary design review* - PDR), e a revisão crítica de modelagem (*critical design review* - CDR) são realizadas nos principais marcos SDLC e criam uma base para as fases subsequentes. A revisão de aptidão para o teste (*test readiness review* - TRR) é completada antes do início do aceite ou do teste de usuário (HORCH, 2003, p. 56).

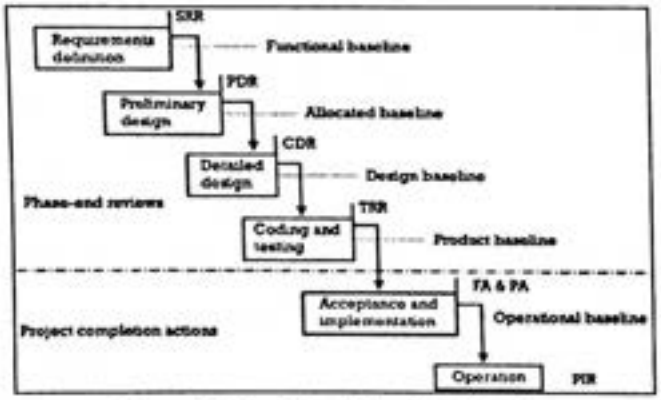

Figura 28- Revisões típicas de fim de fase
FONTE: HORCH, 2003, p. 56

Também são mostradas as auditorias de fim de projeto (*project completion*), conduzidas para determinar a habilitação à entrega, e a revisão pós-implementação, que avalia o sucesso do projeto depois de um período de uso real (HORCH, 2003, p. 57).

A Tabela 19 apresenta os assuntos típicos de cada uma das quatro principais revisões de fim de fase. Os documentos listados são requeridos e considerados o conjunto mínimo aceitável necessário para um desenvolvimento e manutenção de *software* bem sucedido (HORCH, 2003, p. 57).

Tabela 19 – Documentos alvo das revisões de fim de fase [19]

Revisão	Requeridos	Outros
Requisitos de *Software*	*Software* requirements specification *Software* test plan *Software* development plan Quality system plan CM plan Standards and procedures Cost/schedule status report	Interface requirements specification
Modelagem preliminar	*Software* top-level *design* *Software* test description Cost/schedule status report	Interface *design* Database *design*
Modelagem crítica	*Software* detailed *design* Cost/schedule status report	Interface *design* Database *design*
Habilitação para o teste	*Software* product specification *Software* test procedures Cost/schedule status report	User's manual Operator's manual

FONTE: HORCH, 2003, p. 57

A Figura 28 mostra duas análises de fim de projeto (*project completion*) - auditoria funcional (*functional audit* - FA) e a auditoria física (*physical audit* - PA) - que ocorrem ao final do SDLC e que são as análises finais do produto *software* a ser entregue contra seus requisitos aprovados e documentação, respectivamente.

A FA compara o sistema de *software* sendo entregue contra os requisitos aprovados para o sistema. A PA procura garantir que todo o conjunto de "produtos entregáveis" (*deliverables*) possui consistência interna (como por exemplo, o manual do usuário está correto e compatível com a versão do *software* em questão) (HORCH, 2003, p. 58).

[19] Itens da tabela não traduzidos para não ocorrer a perda de sentido em alguns casos, visto que tais terminologias são as reconhecidas no mercado.

A revisão pós-implementação (*post implementation review* - PIR) ocorre uma vez que o *software* esteja em produção, geralmente de 6 a 9 meses após o início de sua utilização, com o propósito de determinar se o *software*, de fato, atendeu às expectativas do usuário em operação real. Os dados coletados pelo PIR ajudam os profissionais da qualidade a melhorarem o processo de desenvolvimento de *software*. A Tabela 20 apresenta algumas das características do PIR (HORCH, 2003, p. 58).

Tabela 20 – Características da revisão pós-implementação (PIR)

Momento	3 a 6 meses após a implementação do *software*
Objetivos do sistema de *Software* versus a experiência	Retorno sobre o investimento Resultados de cronograma Resposta do usuário Históricodo defeito
Utilização dos resultados da revisão	Entrada no processo de análise e melhoria Muitas vezes é ignorado

FONTE: HORCH, 2003, p. 58

As revisões são conduzidas ao longo do SDLC com as de desenvolvimento focando no código e nos seus produtos relacionados (HORCH, 2003, p. 58). As revisões de modelo (*design*) e código que ocorrem durante o curso de várias fases são geralmente na forma de *walk-throughs* e inspeções, de forma a se iniciar logo a eliminação de defeitos no produto sendo examinado. Elas são geralmente informais, o eu as torna mais produtivas e menos ameaçadoras aos egos e sentimentos dos desenvolvedores dos produtos sendo revisados (HORCH, 2003, p. 58). As revisões de teste são análogas às revisões de código, cobrindo os produtos do programa de testes ao invés dos códigos do *software*. Elas incluem os mesmos tipos de revisões formais e informais e ocorrem durante o SDLC. Sua função é examinar o programa de testes enquanto ele está sendo desenvolvido para se certificar que os mesmos exercitarão o *software* de forma a encontrar defeitos e demonstrar que ele atende aos requisitos (HORCH, 2003, p. 58).

2.5.5 Revisões de documentação

Existem inúmeros tipos de revisões de documentação, tanto formais e informais, que são aplicáveis para cada um dos documentos do *software* (HORCH, 2003, p. 59).

A mais básica das revisões é o *walk-through* em par. Outra revisão básica de documentação é a revisão ou auditoria de formato. Uma abordagem mais formal à auditoria de formato é utilizada quando não existe uma revisão de conteúdo agendada (HORCH, 2003, p. 59). Um outro tipo de revisão ainda é a análise de algoritmo, que examina as abordagens específicas que serão utilizadas como solução para os problemas que deverão ser endereçados pelo *software*. As análises de algoritmo são geralmente restritas a sistemas muito grandes e críticos por se tratar de uma atividade custosa em termos de tempo e recursos, tais como *software* para controle de mísseis, transferência eletrônica de fundos, algoritmos de segurança (HORCH, 2003, p. 59).

As revisões de requisitos procuram mostrar que o problema a ser resolvido está completamente descrito. Revisões informais são realizadas durante a preparação do documento. Uma revisão formal é apropriada antes da entrega do documento (HORCH, 2003, p. 59).

As revisões de modelagem (*design*) verificam se o modelo em questão está correto e é rastreável (*traceable back*) em relação aos requisitos aprovados (HORCH, 2003, p. 60). A documentação de teste é revisada para garantir que o programa de testes irá encontrar os defeitos e testará o *software* em relação aos seus requisitos (HORCH, 2003, p. 60).

A documentação do usuário deve não somente apresentar informações sobre o sistema, como também estas devem ser significativas para o leitor. As revisões de documentação do usuário procuram determinar se a documentação atende a alguns critérios como completude, conformidade, e *readability*. A preocupação principal será avaliar se o documento atende a necessidade do usuário de entender como fazer o sistema executar suas funções (HORCH, 2003, p. 60).

Outros documentos são freqüentemente produzidos durante o SDLC e devem ser revisados a medida que eles forem preparados, tal como o plano de desenvolvimento de *software* (*software* development plan), plano do sistema de qualidade de *software* (*software* quality system plan), plano de gerência de mudanças (CM plan), e vários outros que podem ser exigidos contratualmente ou pelos padrões da organização. Vários destes documentos são de natureza administrativa e são preparados antes do início do desenvolvimento de *software* (HORCH, 2003, p. 60).

O autor chama a atenção para o fato de muitas vezes os documentos não receberem a manutenção adequada enquanto o sistema em operação recebe. Isto leva a uma maior dificuldade e custo de manutenção mais tarde. Ignorar a manutenção da documentação resultará em tempo sendo desperdiçado na reinvenção ou na reengenharia da documentação cada vez que uma manutenção do sistema for necessária (HORCH, 2003, p. 60).

2.6 Testes

Segundo Tsao *et al* (2003), o teste de *software* é uma das mais importantes fases na engenharia de *software*, e possui um papel cruscial na qualidade de *software*. No entanto existe muita confusão entre teste e garantia da qualidade de *software*. O autor afirma que o teste não é a garantia da qualidade, pois o teste ajuda a garantir que o produto atenda aos requisitos, enquanto a Garantia da Qualidade endereça atividades ligadas à prevenção de defeitos bem como a remoção daqueles defeitos que se arrastam no produto, incluindo a seleção de normas, a definição de tipos de documentos que devem ser utilizados, os processo que irão guiar as atividades de projeto e as métricas para quantificar os resultados de decisões.

De Young e Rätzmann (2003, p. 25) afirmam que os testes sozinhos não aumentam a qualidade do *software*.

Com relação a atividade em si, Burstein (2003, p. 56) salienta que o teste é uma atividade criativa e desafiadora.

Hutcheson (2003, p. 44) menciona que tradicionalmente o teste tem sido uma ferramenta para a medida da qualidade do produto, no entanto, hoje, o teste precisa estar apto a fazer mais do que simplesmente aferir. É preciso ter a capacidade de adicionar valor ao produto.

Hutcheson (2003, p. 46) afirma que uma maneira de demonstrar o valor dos métodos de teste é medir o benefício financeiro de não utilizar tais métodos e comparar benefício financeiro de utilizá-los. Como exemplo ele menciona um estudo realizado no início da década de 90, onde um grupo de testadores treinados encontraram em uma mesma aplicação uma taxa média de 2

bugs por dia, enquanto que outro grupo de testadores não treinados encontraram uma taxa de 3 *bugs* por dia na mesma aplicação. Depois, quase 90% dos *bugs* encontrados pelos testadores treinados foram fixados, enquanto apenas metade dos *bugs* reportado pelos testadores não treinados foram retornados pelos desenvolvedores como irreproduzíveis (HUTCHESON, 2003, p. 46-47).

Neste estudo os testadores treinados foram remunerados em quase o dobro que os do segundo grupo. Mesmo assim, cada *bug* custou em média US$ 13 para os testadores do primeiro grupo encontrarem um *bug*, enquanto este número foi de US$ 50 para os testadores do segundo grupo encontrarem o mesmo *bug*. Mesmo em se levando em conta os custos de educação em testes, os pertencentes ao primeiro grupo são de longe os mais eficientes (HUTCHESON, 2003, p. 47).

Hutcheson (2003, p. 47) afirma ainda que a melhor razão para se utilizar métodos de teste é que eles trazem às empresa vantagem competitiva, já que elas obtêm um melhor trabalho por um custo menor.

A melhor abordagem para testar *software* é composta por uma mistura de diversas técnicas para Hutcheson (2003, p. 74): a precisão do engenheiro e os benefícios do artista. O desafio do testador é atingir o balanceamento correto entre a liberdade criativa e a disciplina da engenharia, pois segundo o autor, a ciência da engenharia provê métodos que possuem boa aderência às necessidades do desenvolvimento de *software* e testes, no entanto deve-se tomar cuidado para que estes não se tornem rígidos e não imutáveis ou eles deixarão de ser benéficos.

Beizer (1990, p.1) afirma que o teste consome pelo menos metade do trabalho gasto na produção do *software*. O autor ressalva que os números variam, aparentemente por causa das diferentes formas de contabilizar estes esforços. Se forem incluídas todas as atividades relacionadas ao teste, como o *debug*, por exemplo, o tempo total gasto varia entre 30 e 90%. Se somente forem contabilizados os testes conduzidos por um grupo de testes independente, a variação fica entre 10% e 25% (BEIZER, 1990, p. 1).

O autor lembra o fato de que poucos programadores gostam de testar e um menor número ainda gosta de planejar os testes, talvez por se tratar de algo mais abstrato que a programação em si (BEIZER, 1990, p. 1).

Outro problema, segundo Beizer (1990, p.1), e mais grave, é a razão pela qual o teste é feito – encontrar *bugs*. Existe um mito que se os programadores realmente forem bons, não deveriam haver *bugs*. No entanto, as estatísticas mostram que um programa, bem feito, possui ainda de um a três *bugs* a cada 100 linhas de código. O autor ressalva que existem grandes variações neste número médio de 1%, que podem ir de 0.01% a 10%, o que pode ser explicado pela complexidade e a circunstância em questão (BEIZER, 1990, p. 1).

2.6.1 Definição e objetivos do teste

Hutcheson (2003, p. 133) afirma que não existe uma definição única para a palavra testes.

Em 1979, Glenford J. Myers, em seu livro "A arte do teste de software", definiu o teste de *software* como um processo, ou uma série de processos, desenhados para certificar que o *software* faz o que foi planejado para ele fazer, bem como não faz nada que não seja desejado. Para Tsao *et al* (2003, p. 69-72) o teste de *software* é a avaliação dos produtos criados durante os esforços de desenvolvimento de *software*, ou seja, não se restringindo apenas em verificar se o *software* atende aos seus requisitos. Informalmente, o autor Tsao *et al* (2003, p. 69-72), diz que o teste de *software* é o processo de evidenciar os defeitos nos sistemas de *software*. Para Tsao *et al* (2003, p. 69-72), os testes compreendem somente os esforços para encontrar os defeitos e não o *debug* e o reparo dos erros. Tsao *et al* (2003, p. 69-72) afirma ainda que alguns testes estendem o foco para garantir que uma aplicação não faz nada a mais do que era suposto fazer.

Copeland (2004) apresenta a definição formal de testes da norma IEEE *Standard* 610.12-1990 (*IEEE Standard Glossary of Software Engineering Terminology*). De acordo com esta, teste é o processo de operar um sistema ou componente sob condições específicas, observando ou gravando os resultados, e fazendo uma avaliação de alguns aspectos do sistema ou componente. O autor diz ainda que, em linhas gerais, o teste é o processo de comparar "o que é" com "o que deveria ser".

Rick Craig e Stefan Jaskiel propuseram uma definição expandida no livro "*Systematic Software Testing*" (apud Copeland (2004)) que descreve o teste como um processo de ciclo de vida concorrente de engenharia, utilização e manutenção do "*testware*" de forma a mensurar e melhorar a qualidade do *software* sendo testado. Esta definição, segundo Copeland (2004), inclui o planejamento, a análise, e o desenho dos casos de teste adicionalmente ao foco do IEEE na execução dos testes.

Rios e Filho (2003, p. 8) descrevem que o teste é qualquer atividade, que a partir da avaliação de um atributo ou capacidade de um programa ou sistema, seja possíel determinar se ele alcança os resultados desejados.

Quanto aos objetivos do teste, Beizer (1990, p. 2) diz que o primeiro deles é a prevenção aos *bugs*, e que a extensão deste objetivo é a descoberta dos sintomas causados por eles, bem como prover um diagnóstico claro para que eles possam ser corrigidos. Quando se previne um *bug*, há economia de uma série de trabalho pela frente. Por esta razão, Dave Gelperin e Bill Hetzel (apud Beizer, 1990, p. 2) advogam "testar, e então codificar", onde a atividade ideal de testes seria a total prevenção de *bugs* durante o planejamento do teste. No entanto, o autor afirma que este este objetivo é inatingível por completo, já que a atividade é executada por seres humanos. Desta forma, o segundo objetivo é a descoberta de *bugs*, que nem sempre são óbvios e ainda diferentes *bugs* podem se manifestar da mesma forma, ou ainda, um *bug* pode ter vários sintomas (BEIZER, 1990, p. 2).

Craig e Jaskiel (2002) mencionam que o teste preventivo utiliza a filosofia de que o teste pode melhorar a qualidade do *software* se ele ocorrer cedo o suficiente no ciclo de vida. Isto ocorre empregando um conceito muito simples: o processo de escrever casos de teste para testar um requisito (antes do modelagem ou codificação estarem completos) pode identificar falhas nas especificações de requisitos. Encontrando estes problemas o suficientemente cedo, sua correção é uma tarefa relativamente simples e pouco onerosa (CRAIG; JASKIEL, 2002).

Um outro benefício trazido pela criação dos casos de teste antes da codificação, de acordo com Craig e Jaskiel (2002), é que os casos de teste por si mesmos ajudam a documentar o *software* (CRAIG; JASKIEL, 2002).

Broekman e Notenboom (2003, p.1) definem teste como um processo centrado em torno do objetivo de encontrar erros em um sistema, seja para *debug* ou para aceite. Para o autor, o teste ajuda a melhorar a qualidade do sistema, no entanto ele não faz isto diretamente e sim de forma indireta provendo um percepção das fraquezas observadas do sistema e os riscos associados para a organização, permitindo com que os gerentes tomem decisões mais bem informados sobre como alocar recursos para melhorar a qualidade dos sistemas. (BROEKMAN; NOTENBOOM, 2003, p. 1)

Os autores Li e Wu (2004) e Horch (2003) concordam com o objetivo do teste de encontrar erros, bem como complementam este objetivo em verificar se o *software* atende aos requisitos da forma como é percebido pelo usuário. Horch (2003) complementa ainda dizendo que considera uma perda de tempo os testes que não sejam baseados em verificar a adequação aos requisitos.

Um outra definição é apresentada em Burnstein (2003, p. 27, 30 e 64), onde o teste é definido como o processo de exercitar um *software* utilizando um conjunto de casos de teste selecionados com a intenção de revelar os defeitos e avaliar sua qualidade.

Beizer (1990, p. 2-3) descreve uma evolução na vida mental do testador, o que muda a sua percepção sobre os objetivos do teste, caracterizada pelas seguintes fases (BEIZER, 1990, p. 2-3):

- Fase 0 – não existe diferença entre testar e realizar o *debug*. Não existe outro objetivo para o teste além de suporte ao *debug*.
- Fase 1 – o objetivo do teste é mostrar que o *software* funciona.
- Fase 2 – o objetivo do teste é mostrar que o *software* não funciona.
- Fase 3 – o objetivo do teste não é provar nada, mas reduzir o risco percebido do não funcionamento à um valor aceitável.
- Fase 4 – o teste não é um ato, e sim uma disciplina mental que resulta em *software* de baixo risco sem muito esforço de testes.

Beizer (1990, p. 3) aproveita para separar bem dois conceitos que são muito confundidos - teste e *debug* – afirmando que eles não são sinônimos. A proposta do primeiro é mostrar que um programa tem *bugs*, enquanto o segundo objetiva encontrar erros que causam a falha do

programa, bem como planejar e implementar as mudanças no *software* que corrigem o erro. O *debug* geralmente vem logo após o teste, mas eles diferem em objetivos, métodos e, de acordo com o autor, mais importante ainda, em termos psicológicos (BEIZER, 1990, p. 3).

2.6.2 Ciclo de vida dos testes

Já que os custos de encontrar e corrigir um defeito cresce dramaticamente com a extensão de tempo no qual o defeito está presente, especialmente nos casos de modelagem e requisitos, quando muitas vezes as falhas somente são descobertas em aceite ou na operação, fazendo com que o *software* volte grande parte do ciclo de desenvolvimento, uma alternativa empregada é a execução de um programa de testes ao longo de todo o ciclo de desenvolvimento de *software* composto por vários níveis (HORCH, 2003, p. 73).

O programa de testes se inicia na fase de requisitos e, efetivamente, nunca termina, pois testes de regressão são realizados cada vez que o sistema sofre alterações. A Figura 29 mostra este conceito (HORCH, 2003, p. 73).

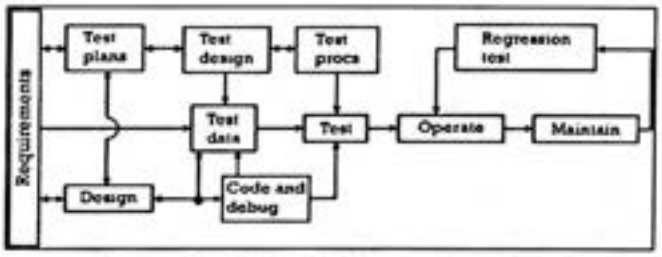

Figura 29- Ciclo de vida de teste de *software*
FONTE: HORCH, 2003, p. 73

Os testes seguem o progresso natural do ciclo de vida do *software* (SLC). A medida que o teste progride, a ênfase vai mudando de simplesmente encontrar o defeito para uma demonstração mais sofisticada dos requisitos. No início, o teste procuta exercitar o máximo de caminhos no *software*, de forma a encontrar erros na codificação, na modelagem, e assim por diante. A medida que o ciclo de vida de desenvolvimento de *software* (SDLC) amadurece, há menos oportunidades de exercitar todos os caminhos no *software*, de forma que a concentração passa a ser na integração dos módulos e subsistemas em um todo, e exercitar este todo em relação aos requisitos. O objetivo básico de encontrar defeitos permanece, no

entanto existe confiança nos testes anteriores para os detalhes mais internos de cada módulo. Mais tarde, o teste lida com o sistema propriamente dito e com suas interfaces com o "mundo" externo, de acordo com o que foi definido nos requisitos (HORCH, 2003, p. 73).

Horch (2003, p. 73) ressalta que todos os testes são planejados para validar a implementação do *software* contra os requisitos aprovados.

O ciclo de vida dos testes apresentados por diversos autores varia muito pouco em torno das seguintes etapas: teste unitário (HORCH, 2003, p. 73-74; BEIZER, 1990, p. 20-22; COPELAND, 2004), teste de modulo (HORCH, 2003, p. 73-74) ou de componente (BEIZER, 1990, p. 20-22), teste de integração (HORCH, 2003, p. 73-74; BEIZER, 1990, p. 20-22; COPELAND, 2004), teste de sistema (BEIZER, 1990, p. 20-22; COPELAND, 2004), teste de usuário (HORCH, 2003, p. 73-74) ou de aceite (HORCH, 2003, p. 73-74; COPELAND, 2004), e teste de regressão (HORCH, 2003, p. 73-74).

O teste unitário é aquele realizado para mostrar que uma unidade não satisfaz suas especificações funcionais ou sua estrutura não está de acordo com o planejado, onde uma unidade é o menor pedaço testável de *software*, geralmente resultado do trabalho de um programador apenas, e que consiste em poucas linhas de código (BEIZER, 1990, p. 21; COPELAND, 2004).

Tsao *et al* (2003, p. 117-154) define dois principais conjuntos de métodos para os testes unitários ou de módulo - caixa preta e caixa branca. O primeiro foca na validação das características funcionais e comportamentos requeridos nos módulos do *software* de uma ótica externa. Estes métodos de testes são desenvolvidos para gerar testes baseados na especificação de um determinado módulo. Já o segundo, de caixa branca, foca na validação das estruturas do programa, comportamentos e lógicas dos módulos sob uma ótica interna. Estes testes são planejados baseados no código fonte de um determinado módulo. Ou seja, o teste de caixa branca dos módulos utiliza testes baseados no programa para descobrir erros na lógica interna, dados e estruturas, enquanto o teste de caixa preta dos módulos utiliza verificações baseadas em requisitos para descobrir os erros nas funções externas, comportamentos e interfaces.

O teste de componente é realizado para mostrar que o componente não satisfaz sua especificação funcional ou sua estrutura não está de acordo com o planejado, onde um componente é um agregado de uma ou mais unidades (BEIZER, 1990, p. 21).

O teste de integração é o processo pelo qual componentes são agregados para criar outros ainda maiores. Desta forma, o teste de integração visa mostrar que mesmo com os componentes individualmente sendo satisfatórios, de acordo com o que foi demonstrado pelo teste de componente, a combinação deles está incorreta ou inconsistente. Ou seja, este teste busca expor os problemas que surgem com a combinação de componentes (BEIZER, 1990, p. 21; COPELAND, 2004).

O teste de sistema busca encontrar erros que não podem ser atribuídos a componentes bem como inconsistências entre os componentes, ou entre as interações planejadas dos componentes e outros objetos. Este teste busca encontrar problemas e comportamentos que somente podem ser expostos através do sistema inteiramente integrado ou uma maior parte dele (BEIZER, 1990, p. 22; COPELAND, 2004). Um sistema pode ser considerado um grande componente (BEIZER, 1990, p. 22) de tudo o que torna o produto apto a ser entregue ao cliente. Os testes de sistema incluem testes de performance (BEIZER, 1990, p. 22; COPELAND, 2004), seguranca (BEIZER, 1990, p. 22; COPELAND, 2004), configuração (BEIZER, 1990, p. 22), recuperação (BEIZER, 1990, p. 22; COPELAND, 2004), confiabilidade (COPELAND, 2004), funcionalidade (COPELAND, 2004), usabilidade (COPELAND, 2004).

O teste de aceite é aquele que quando completado com sucesso, irá resultar no cliente aceitando o *software* e realizando o pagamento do mesmo (COPELAND, 2004).

2.6.3 A evolução do teste

Craig e Jaskiel (2002) apresentam uma evolução da definição da definição dos testes ao longo dos anos na Tabela 21.

Tabela 21 – Definições dos testes ao longo dos anos

Ano	Definição
1979	Teste é o processo de executar um programa ou sistema com a intenção de encontrar erros.
1983	Teste é qualquer atividade para avaliar um atributo de um programa ou sistema. O teste é a medida da qualidade do *software*.
2002	Teste é o ciclo de vida concorrente de engenharia, utilização e manutenção do *testware* de forma a mensurar e melhorar a qualidade do *software* sendo testado.

FONTE: CRAIG e JASKIEL, 2002

Craig e Jaskiel (2002) mencionam que a primeira definição, de Glenford Myers em 1979, trazida em seu clássico *The Art of Software Testing*, provavelmente era a melhor definição disponível na época e mostra o pensamento nela – testar o *software* ao final do ciclo de desenvolvimento com o principal objetivo de encontrar erros. No entanto, como defende na atualidade Burnstein (2003, p. 32) em um dos princípios em que cita, as atividades de teste devem estar integradas ao ciclo de vida do *software* (CRAIG; JASKIEL, 2002).

Já em 1983, a definição mudou com a inclusão da avaliação da qualidade do *software* ao invés de simplesmente encontrar erros. Em *The Complete Guide to Software Testing*, Bill Hetzel (apud, Craig; Jaskiel, 2002) diz que o teste é qualquer atividade para avaliar um atributo de um programa ou sistema, bem como é a medição da qualidade do *software*. Esta definição continua válida, entretanto, possui um problema em relação a seu escopo e por isto Craig e Jaskiel (2002) apresentam e adotam outra definição, a de que o teste é o ciclo de vida concorrente de engenharia, utilização e manutenção do testware de forma a mensurar e melhorar a qualidade do software sendo testado, tal como já mencionado anteriormente. (CRAIG; JASKIEL, 2002).

Segundo Horch (2003, p. 87), até recentemente, a preferência sobre quem deve realizar os testes recaía em testadores independentes, tal como defende Burnstein (2003, p. 30), por exemplo. Embora para Horch (2003, p. 87) isto ainda continue válido em algumas situações, a definição de independente tem sido trocada em muitas situações.

A Tabela 22 sugere os testadores apropriados para cada tipo de teste de acordo com Horch (2003, p. 87)

Tabela 22 - Responsabilidade dos testes

Tipo de Teste	Testador
Debug	Programador
Teste unitário	Programador
Teste de módulo ou objeto	Programador
Teste de integração de módulo ou objeto	Terceiros
Teste de integração de sistema ou subsistema	Terceiros
Teste de sistema	Time de teste de desenvolvimento
Teste de aceite/usuário	Time de teste de usuários

FONTE: HORCH, 2003, p. 87

Dado que os testes unitários, *debug*, testes de módulo e de integração são tarefas da equipe de desenvolvimento, esta deve ser responsável pela qualidade de seu próprio trabalho, de acordo com Horch (2003, p. 88). A partir do momento em que ocorre uma integração em larga escala e o teste de sistema deve ser realizado, um time de teste independente dos produtores deve realizar os testes, já que, como o objetivo do teste é encontrar erros, a objetividade de um time de testes independente irá melhorar muito a qualidade do teste. Já os testes de usuário ou de aceite devem ser conduzidos pelos usuários propriamente ditos, segundo Horch (2003, p. 88), de preferência no ambiente do usuário, ajudando assim a assegurar que o *software* atenda às expectativas deles, bem como os requisitos oficialmente aprovados (HORCH, 2003, p. 88).

Já no caso dos testes de regressão, Horch (2003, p. 88) diz que ele é conduzido por diferentes pessoas envolvidas no ciclo de vida do *software*, já que os desenvolvedores farão os testes de regressão dos módulos, enquanto os times independentes de teste farão os testes de regressão das integrações e do sistema.

2.6.4 Os paradigmas do teste

Copeland (2004) apresenta os paradigmas atuais do teste de *software* – testes baseados em *scripts* e testes exploratórios. Enquanto o primeiro, de acordo com o autor, é baseado em um exame sequencial dos requisitos, e no desenho e documentação dos casos de teste, seguido pela execução destes, dentro de uma filosofia "Planeje seu trabalho, trabalhe o seu plano" (*Plan your work, work your plan*), o segundo enfatiza o aprendizado, o desenho e a execução do teste simultaneamente ao invés de uma abordagem sequencial. Como o próprio autor

observa, cada um destes paradigmas em sua obra são descritos de uma forma extremada, de uma maneira inflexível na qual raramente qualquer um deles é utilizado na prática, já que com muita frequência um teste baseado em roteiro pode ser um tanto exploratório, tal como um teste exploratório pode ser um tanto baseado em roteiro (COPELAND, 2004).

O teste baseado em *scripts* enfatiza o valor de planejar com antecedência os testes como um método de detectar defeitos de requisitos e modelagem antes da escrita do código e da colocação do sistema em produção. Ele é focado na contabilidade (*accountability*) e repetibilidade (COPELAND, 2004). Dentro desta filosofia, Burnstein (2003, p. 26-33) apresenta uma série de princípios de testes, dentre os quais defende que os testes devem ser repetíveis e reutilizáveis, bem como devem ser planejados.

Já o teste exploratório desafia a idéia de que os testes devem ser planejados no início do projeto, quando o conhecimento a respeito do produto é geralmente mínimo. Seu foco está no aprendizado e na adaptabilidade (COPELAND, 2004).

As vantagens do teste baseado em roteiro incluem a documentação formal, cobertura e a rastreabilidade. Nos testes exploratórios, os testadores planejam e executam os testes enquanto exploram o produto. Ele é vital nas situações em que a escolha dos próximos casos de teste a serem executados não podem ser feita antecipadamente pois eles devem ser escolhidos baseado nos testes anteriores e em seus resultados (COPELAND, 2004).

Copeland (2004) ressalta ainda que o uso dos testes baseados em roteiro não impede o uso de testes exploratórios e vice-versa, de forma que os testadores mais espertos devem utilizar as ferramentas que forem necessárias para cumprir seus objetivos (COPELAND, 2004).

De Young e Rätzmann (2003, p. 14) apresentam uma técnica exploratória e de teste integrado denominada Teste Rápido de Aplicação (*Rapid Application Testing*), cujo objetivo é encontrar os defeitos mais sérios no *software* o mais rápido possível. Esta estratégia provê uma série de técnicas que auxiliam na busca deste objetivo. O autor define ainda que o teste rápido de aplicação pode ser descrito como a arte de fazer um programa falhar (DE YOUNG; RÄTZMANN, 2003, p. 14).

2.6.5 A barreira da inexequibilidade do teste completo

Diversos autores afirmam que é impossível testar todas as condições possíveis de um *software* (BEIZER, 1990, p. 26; COPELAND, 2004). *Testing Object-Oriented Systems*, Robert Binder apud (COPELAND, 2004) traz um bom exemplo que aponta esta impossibilidade. Um simples programa de 4 linhas, mostrado a seguir, contém apenas uma linha errada, tal como apontado ao lado desta.

```
int blech (int j) {
j = j -1;        // o correto seria j = j + 1
j = j / 30000;
    return j;
  }
```

Se este *software* for implementado em um computador de 16 bits ele poderá aceitar 65536 entradas e, portanto, seria possível delinear 65536 casos de teste (COPELAND, 2004).

A Tabela 23 mostra 4 casos de teste selecionados aleatoriamente, bem como o resultado esperado se o *software* estivesse correto, bem como o resultado real obtido com sua implementação incorreta (COPELAND, 2004).

Tabela 23- Entradas x Saídas Reais x Saídas Esperadas

Entrada (j)	Resultado Esperado	Resultado Real
1	0	0
42	0	0
40000	1	1
-64000	-2	-2

FONTE: COPELAND, 2004

Nota-se que nenhum dos casos de teste revelou o problema, e, conforme Copeland (2004) afirma, apenas 4 casos de teste, dentre 65536 são capazes de revelar a discrepância. Dado que em um *software* deste tamanho e complexidade, dificilmente serão realizados 65536 testes, qual é a probabilidade de selecionar um ou mais dos 4 casos de teste corretos dentre 65536 possíveis?

Já de acordo com Beizer (1990, p. 26), há três diferentes abordagens que podem ser utilizadas para demonstrar que um programa está correto: os testes baseados na estrutura, baseados na função, e provas de conformidade formais. Cada uma delas permite concluir que o teste completo, na filosofia de prova de que o *software* está totalmente correto, não é, nem teoricamente e nem praticamente, possível. Um teste functional completo consiste em sujeitar o programa a todas as entradas possíveis produzindo comportamentos ou saídas corretas ou incorretas. Uma simples entrada de 10 caracteres possui 2^{80} possíveis combinações de entrada, que se cada uma for testada em um intervalo de tempo de um microsegundo, o teste durará o dobro da idade estimada do universo. Por outro lado, em se tratando de teste estrutural, a idéia é exercitar ao menos uma vez cada caminho possível através das rotinas do *software*. Apenas uma pequena rotina do *software* pode possuir milhões ou bilhões de possíveis caminhos, de forma que um teste de todos os caminhos é impraticável (BEIZER, 1990, p. 26).

As de prova de conformidade são baseadas em uma combinação de conceitos de funcionais e estruturais. Nesta técnica cada instrução do programa é examinada e utilizada em um passo de uma prova indutiva de que a rotina irá produzir a saída correta para qualquer sequência de entradas, de acordo com as especificações. No entanto, sem contar o fato de tais provas serem muito caras e suas restrições de aplicação em apenas casos numéricos, o autor Beizer (1990, p. 26) levanta um questionamento mais fundamental: como pode-se saber se a especificação pode ser provada corretamente? Mesmo em se considerando que isto seja possível, o mecanismo utilizado para provar o *software*, os passos na prova, a lógica utilizada, e assim por diante, deve ser provada. Os matemáticos e lógicos são tão imunes a erros quanto os programadores e os testadores. Isto leva a uma sequência sem fim de considerações não verificáveis. (BEIZER, 1990, p. 26)

Beizer (1990, p. 26) resumiu as barreiras teóricas ao teste completo em:

- Nunca se pode ter certeza que as especificação estão corretas;
- Nenhum sistema de verificação pode verificar todo o programa;
- Nunca se pode ter certeza que o sistema de verificação está correto.

Desta forma, Beizer (1990, p. 26) afirma que o objetivo passa a ser o provimento de testes suficientes para garantir que a probabilidade de falha devido aos *bugs* "hibernantes" seja

baixa o suficiente que torne o *software* aceitável. No entanto, o que é aceitável para um jogo de computador é insuficiente para um *software* de controle de um reator nuclear, por isto, cada tipo de aplicação possui uma demanda diferente por testes devido ao nível de confiabilidade exigido (BEIZER, 1990, p. 26).

Broekman e Notenboom (2003, p. 1) reconhecem a existência de uma regra universal que afirma ser impossível encontrar todos os defeitos de um *software*, já que nunca há recursos suficientes, como tempo, pessoas e dinheiro para testar um *software* completamente. Desta forma, Broekman e Notenboom (2003, p. 1) dizem que escolhas devem ser feitas sobre como distribuir os recursos disponíveis de forma inteligente.

De Young e Rätzmann (2003, p. 13) mencionam que durante muito tempo a base para a geração dos planos de teste e métricas era a cobertura das propriedades funcionais e a cobertura das especificações, no entanto, quando se percebeu que não se pode testar tudo, métodos foram desenvolvidos para descobrir redundâncias nos casos de teste e dados de teste.

2.6.6 Paradoxos do teste de *software*

Beizer (1990, p. 9) apresenta dois paradoxos do teste de *software*. O primeiro deles é uma analogia feita ao fazendeiro que utiliza diferentes pesticidas a cada ano, pois a cada pesticida utilizado, este se torna ineficiente no próximo ano, já que as pragas remanescentes são resistentes aos seus efeitos. Da mesma forma, o autor afirma que cada método utilizando para prevenir ou encontrar erros deixa um conjunto de *bugs* residuais contra os quais estes métodos são ineficientes, de forma que para combater isto, deve-se utilizar uma série de métodos variados. Por outro lado, ele ressalta que isto não é tão ruim, visto que o *software* vai se tornando cada vez melhor (BEIZER, 1990, p. 9).

O outro paradoxo é a barreira da complexidade. Neste, o autor afirma que a complexidade do *software* e dos *bugs* cresce ao limite de nossa habilidade de lidar com esta complexidade. Beizer (1990, p. 9) explica que eliminando-se os *bugs* mais fáceis passa-se para uma outra escala de características e complexidade, só que desta vez tem-se *bugs* mais sutis para lidar, apenas para manter a confiabilidade que se tinha antes. A sociedade deseja sempre algo a mais. Desta forma, os usuários sempre empurram o desenvolvimento para a barreira de

complexidade. O quão perto pode-se aproximar dela é determinado pelas forças das técnicas e a capacidade de utilizá-las contra os *bugs* mais complexos e sutis (BEIZER, 1990, p. 9).

Um princípio muito importante e que pode ajudar a guiar os esforços de teste é apontado por Burnstein (2003, p. 30). Este princípio diz que a probabilidade da existência de defeitos adicionais em um componente de um *software* é proporcional ao número de defeitos já detectados neste componente.

2.6.7 Quando parar de testar

Se por um lado o teste completo é impossível, um questionamento que vem a tona é quando deve-se parar de testar um *software*. Na obra *The Complete Guide to Software Testing*, Bill Hetzel (apud Copeland, 2004) escreveu que o teste termina quando as capacidades do sistema foram mensuradas e ele foi corrigido o suficiente para se ter confiança de que se está pronto para ser submetido ao teste de aceite. Copeland (2004) diz que, embora corretos, os termos "corrigido o suficiente" e "ter confiança" são vagos.

Copeland (2004) descreve ainda que Boris Beizer escreveu que não existe um critério simples, válido e racional de término da etapa de teste. Além disto, dado um conjunto qualquer de critérios aplicáveis, o quão exato cada um é ponderado depende muito do produto, do ambiente, da cultura e da atitude ao risco. Mais uma vez, Copeland (2004) faz a ressalva de que isto não ajuda muito a saber quando se deve parar de testar.

Copeland (2004) afirma que embora Beizer diga que não exista um simples critério para parar os testes, muitas organizações adotaram um de alguma forma. Os cinco critérios básicos frequentemente utilizados para decidir quando parar de testar são (COPELAND, 2004):

- os objetivos de cobertura previamente definidos foram atingidos;
- a taxa de descoberta de defeitos está abaixo de um nível previamente definido;
- o custo marginal de encontrar o próximo defeito excede a perda esperada com aquele defeito;
- o time de projeto chega a um concenso de que é o momento de lançar o produto;
- o chefe diz que é a hora de finalizar os testes.

A obra (BEIZER, 1990) descreve um procedimento para conduzir e parar os testes:

a) Inicialmente deve-se listar as piores situações que o *software* pode trazer. Deve-se listá-las em termos dos sintomas que elas produzem e como o usuário lidará com eles.
b) As conseqüências dos sintomas devem ser convertidas em custos, que geralmente são aqueles associados à mão-de-obra para corrigir os problemas, ou, no caso da extensão ao público, pode ser o custo judicial, negócios perdidos, etc.
c) Deve-se ordenar a lista dos maiores para os menores custos e descartar aqueles com os quais pode-se conviver.
d) Baseando-se na experiência, dados medidos, intuição e estatísticas publicadas, deve-se postular os tipos de *bugs* que provavelmente criarão os sintomas de cada situação.
e) Para cada situação, foi desenvolvida uma lista de possíveis *bugs* causadores. Deve-se então ordenar a lista em ordem decrescente de probabilidade, onde esta foi julgada baseando-se em estatísticas, experiência, intuição etc. O mesmo tipo de erro irá aparecer em diferentes situações. A importância de um tipo de erro é calculada multiplicando-se o custo esperado da situação pela probabilidade do *bug* e somando isto ao longo de todas as situações onde ele aparece.
f) Deve-se então ordenar os tipos de *bug* em ordem decrescente de importância.
g) Os testes e os processos de inspeção de garantia da qualidade devem ser desenhados utilizando-se os métodos que são mais eficientes contra os *bugs* mais importantes.
h) Se um teste foi bem sucedido, então alguns pesadelos ou partes deles foram embora. Se um teste falhou, então o pesadelo é possível, mas corrigindo-se o erro, ele também se vai. O teste, portanto, traz informações que podem ser utilizadas para revisar suas probabilidades de pesadelos estimadas. A medida que os testes são realizados, deve-se revisar as probabilidades e reordenar a lista de pesadelos, bem como eventualmente revisar a estratégia de teste.
i) Deve-se parar de testar quando a probabilidade de todos os pesadelos forem mostradas inconseqüentes como um resultado de forte evidência produzida pelo teste.

2.6.8 Os motivos da dificuldade do teste de *software*

Fewster e Graham (1999) afirmam que hoje o teste de *software* está mais difícil. Isto se deve à variedade de sistemas operacionais, linguagens de programação e plataformas de máquinas, bem como pelo fato de, nos tempos atuais, a maior parte das atividades estarem apoiadas de alguma forma na utilização de computadores, diferentemente da década de 70.

Segundo Craig e Jaskiel (2002) outros problemas que tornam os testes de *software* tão difíceis são os requisitos incorretos e ambíguos bem como os cronogramas muito apertados, sem contar que muitas vezes os testadores não estão adequadamente treinados.

Hutcheson (2003, p. 130) apresenta o resultado de um estudo que quantifica esta dificuldade, bem como descreve dois motivos principais, em sua visão, desta situação. A obra mostra que, de acordo com o resultado de um estudo conduzido em 1993 e 1994, de 80 a 90% dos *bugs* encontrados em produção foram encontrados durante o teste e que aproximadamente 60% destes erros foram considerados "difícieis de reproduzir". A cobertura dos testes foi suficiente, no entanto o esforço de testes não, já que não haviam recursos suficientes para rastrear estes erros e corrigí-los. O autor afirma ainda que os esforços de teste geralmente não falham porque os *bugs* não foram descobertos e sim pelo fato de que um número inaceitável de *bugs* permanece no produto. O autor diz que um dos motivos de alguns *bugs* serem difíceis de se reproduzir é que eles existem somente em certos ambientes, ou seja, ele pode não estar presente no *software* que está sendo testado e sim em outro *software* que suporta o ambiente, como o sistema operacional por exemplo, ou um sistema de gerenciamento de banco de dados. Outra razão apresentada pelo autor, pela qual alguns *bugs* são difíceis de se reproduzir, é que os sistemas onde o *software* está rodando são grandes e com muitos usuários, de forma que não podem ser considerados uma máquina de estados finitos. Pois enquanto o testador está realizando os testes, outras aplicações podem estar sendo utilizadas simultaneamente, como por exemplo um CD pode estar sendo tocado, bem como uma busca na internet por algum documento pode estar sendo realizada, bem como um processador de textos pode estar aberto para a realização de anotações etc. Quando um erro ocorre, consequentemente, recriar os estados exatos no qual se encontrava o sistema no momento em que a falha ocorreu é impossível (HUTCHESON, 2003, p. 130).

2.6.9 A documentação do teste

Horch (2003, p. 67) descreve que a documentação do teste é iniciada durante a fase de requisitos com a preparação dos planos de testes inciais tendo em mente o teste final de aceite, bem como os testes intermediários que irão examinar o *software* durante o desenvolvimento. O autor ressalta a importância de que cada requisito seja endereçado no plano total de teste. A medida que ocorre a evolução das fases do desenvolvimento, o conjunto de documentação de teste cresce, e, deve se manter rastreável em direção aos requisitos. A documentação do programa de teste deve também refletir as mudanças evolutivas nos requisitos a medida que eles ocorrem, na opinião do autor. A Figura 30 mostra como os requisitos podem ou não ser propriamente endereçados pelos testes. Algum deles podem se perder, tal como alguns teste podem aparecer. Uma revisão apropriada dos planos de teste pode ajudar a identificar estes desvios (HORCH, 2003, p. 67).

A medida que cada fase do ciclo de vida do desenvolvimento de *software* progride, partes adicionais do programa de teste são desenvolvidas. Os casos de teste com seus dados de teste são preparados, bem como os cenários de teste e os procedimentos de teste a serem executados (FONTE: HORCH, 2003, p. 68).

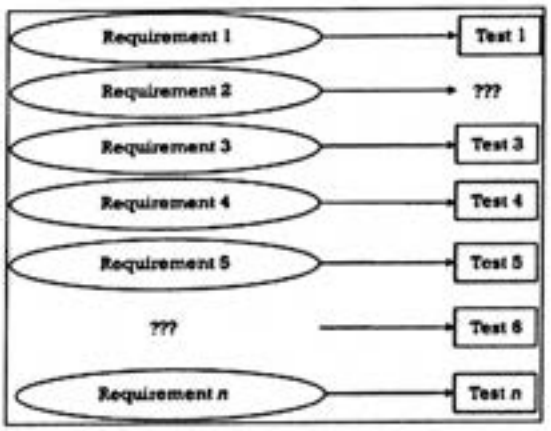

Figura 30- Confrontando os testes com os requisitos
FONTE: HORCH, 2003, p. 67

Com relação ao desenvolvimento dos casos de testes, há algumas observações que podem ser encontradas em Burnstein (2003, p. 27-33). Uma delas é que eles devem ser desenvolvidos

tanto para condições de entrada válidas como inválidas. Outra observação é, em se considerando como o objetivo do teste encontrar defeitos, um bom caso de teste é aquele que tem uma elevada probabilidade de revelar defeitos ainda não detectados (BURNSTEIN, 2003, p. 27-33).

Para cada teste, um critério passou/falhou é determinado, baseado nos resultados experados de cada caso de teste ou cenário. Em cima disto, os relatórios de teste são preparados para guardar os resultados do esforço de teste (HORCH, 2003, p. 68).

O teste de aceite não é o único teste com o qual a documentação de teste se preocupa. Existem também os testes unitários, de módulos, de integração, e de subsistemas, que devem ser planejados e documentados, pois são parte do planejamento total de teste e processo de desenvolvimento (HORCH, 2003, p. 68).

Segundo descreve Horch (2003, p. 4), os relatórios de teste documentam os resultados reais dos esforços de teste a medida que ele progride. Para cada teste executado, um relatório de resultados esperados, reais, e as conclusões dos testes deve ser preparado, incluindo as anomalias, erros e eventualmente resultados questionáveis. Uma vez estes pontos sejam corrigidos, os testes são re-executados para mostrar que os defeitos foram corrigidos. O autor afirma ainda que a medida que o teste progride, o nível de detalhes dos relatórios de teste aumenta.

2.6.10 As ferramentas de automação e de apoio aos testes

Durante muito tempo, o teste de *software* foi realizado manualmente, ou seja, um testador humano executando a aplicação utilizando processos pré-definidos. Desde os primórdios das indúsria de *software*, os engenheiros de *software* têm feito grandes esforços para automatizar os processos de teste. Muitas empresas bem sucedidas desenvolveram ferramentas de teste que se contram no mercado. Hoje, há muitas ferramentas comerciais que podem ser utilizadas para encontrar erros de forma a repará-los antes da liberação do produto (LI; WU, 2004).

Trata-se de um assunto novo, principalmente no Brasil onde os custos destas ferramentas muitas vezes é proibitivo, até porque a justificativa do retorno sobre o investimento baseado

no custo do profissional no Brasil faz com que este retorno seja duvidoso, ao contrário de países como os EUA ou Europa, onde o custo da mão de obra é elevada e a prática de horas extras de trabalho encarecem demasiadamente as atividades. Mesmo com estas dificuldades, uma série de empresas tem investido em tais tecnologias buscando um melhor resultado em seus testes, um melhor controle em seus processos de garantia de qualidade, e, por fim, o desenvolvimento de um *software* de maior qualidade.

A automação do teste de *hardware* se encontra de 10 a 15 anos na frente da automação de testes de *software*, pelo fato do teste de *hardware* ser mais fácil. (BEIZER, 1990, p. 309-313).

Segundo Dustin (2003), as ferramentas de teste automatizado podem melhorar o esforço de teste desde que as expectativas sejam gerenciadas, as limitações das ferramentas sejam compreendidas, e uma ferramenta compatível com o ambiente do *software* seja selecionado. O autor chama a atenção para a decisão que deve ser feita, entre selecionar uma ferramenta de testes existente no mercado que atenda as necessidades do projeto, desenvolver uma ferramenta para atender as necessidades, ou, fazer os testes manualmente. No contexto desta decisão, Dustin (2003) menciona que as ferramentas disponíveis comercialmente podem, muitas vezes, apresentar uma série de funcionalidades adicionais e não necessárias no projeto, o que pode significar um custo maior. Por outro lado, ele chama atenção para a necessidade de uma análise criteriosa de que uma ferramenta construída não será mais onerosa no longo prazo do que comprar uma. Por fim, o autor reforça que o exame destes assuntos o mais cedo possível no ciclo de vida do desenvolvimento evita mudanças custosas mais tarde (DUSTIN, 2003).

Fewster e Graham (1999) ressaltam que muitas vezes a automação de testes é mais cara do que realizar o teste uma vez manualmente. Para obter os benefícios da automação, segundo os autores, os testes a serem automatizados precisam ser selecionados e implementados cuidadosamente. Eles reforçam ainda que a qualidade da automação é independente da qualidade do teste.

Fewster e Graham (1999) mencionam que a automação não afeta a efetividade do teste, pois, se um teste não traz nenhum resultado, quando automatizado, ele continuará trazendo nenhum resultado, só que mais rápido. Uma vez implementado, o teste automatizado é geralmente muito mais econômico, o custo de execução é uma fração do esforço de executa-lo

manualmente. No entanto, eles ressaltam que os testes automatizados geralmente custam mais para criar e manter. Se não for dada a devida atenção à manutenção durante a automação dos testes, a atualização de uma *suite* de teste automatizada pode custar tanto quanto, senão mais, que a execução manual dos testes (FEWSTER; GRAHAM, 1999).

Fewster e Graham (1999) apresentam um diagrama (Figura 31) com os quatro atributos de qualidade de um caso de teste em um diagrama de Keviat. Um caso de teste executado manualmente é mostrado pelas linhas sólidas. Quando o mesmo teste é automatizado pela primeira vez, ele se tornará menos evoluído e menos econômico (pois foi necessário mais esforço para automatizá-lo). Depois que o teste automatizado foi executado um número de vezes ele se tornará muito mais econômico que o mesmo teste executado manualmente. Para uma *suite* de testes automatizado ser efetiva e eficiente é necessário iniciar com o ingrediente básico de uma boa suite de teste, um conjunto de testes habilidosamente modelados por um testador para exercitar as coisas mais importantes. Então, deve-se aplicar os conhecimentos de automação para automatizar os testes de tal forma que eles sejam criados e mantidos em um patamar de custos razoável (FEWSTER; GRAHAM, 1999).

A pessoa que constrói e mantem os artefatos associados ao uso de uma ferramenta de execução de testes é o Automatizador de Teste. Ele pode, por exemplo, ser um desenvolvedor que auxiliar a equipe de testes nas necessidades de automação. O autor reforça que suas habilidades é que determinarão o quão fácil será adicionar novos testes automatizados, o quão fácil será a manutenção dos mesmos, e quais os benefícios a automação trará (FEWSTER; GRAHAM, 1999).

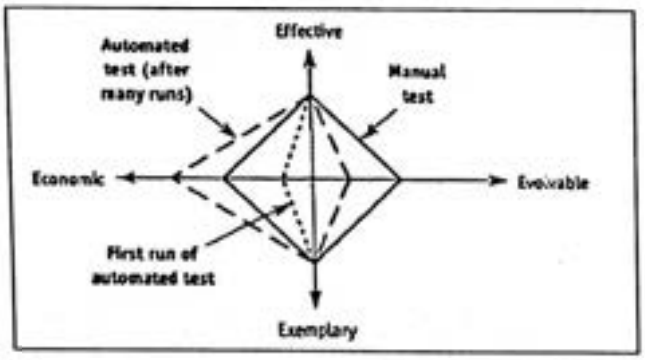

Figura 31 - Diagrama com dos 4 atributos de benefício de um caso de teste
FONTE: FEWSTER e GRAHAM, 1999

Li e Wu (2004, p. 3) mencionam que com uma ferramenta de automação de testes, o teste de *software* pode ter uma economia de tempo, seguir normas, bem como aumentar a qualidade do produto. O autor reforça que o teste de um produto totalmente novo é sempre uma experiência de aprendizado para o testador e que o conhecimento profundo de um projeto é que torna possível a automação total. O autor diz que tecnicamente muitos testes podem ser automatizados, no entanto a decisão do que automatizar além de técnica, é também gerencial.

Uma análise, segundo o autor deve incluir (LI; WU, 2004, p. 3):

- Na sociedade atual, os projetos de *software* são complexos e pretendem resolver problemas complicados. Os fabricantes de ferramentas de teste de *software* comerciais geralmente precisam de tempo para aprender sobre um problema particular e endereçá-los com as tecnologias. De formar a atender a janela de tempo do projeto, o time de teste deve desenvolver uma ferramenta de teste automatizado que seja complementar as ferramentas de testes comerciais.
- Às vezes, os engenheiros de teste precisam determinar se devem incorporar o teste automatizado em projetos existentes ou em um novo. No inicio, os processos de teste sempre envolvem testes manuais. Os engenheiros de teste utilizam isto como um processo de aprendizado. A medida que o projeto progride, os engenheiros de teste se tornam mais conhecedores do produto e os problemas potenciais se tornam mais previsíveis. As ferramentas de automação de teste então podem ser adicionadas para lidar com os possíveis problemas. Eventualmente, esta ferramenta pode beneficiar muito os futuros projetos na organização.
- As organizações que empregam o *Extreme Programming (XP)* não enfatizam a documentação detalhada dos requisitos. Ao invés disto, o código delas é constantemente modificado e atualizado. Os roteiros de teste são parte do controle de código fonte juntamente com o código do programa. Desta forma, o teste automatizado é crítico nas práticas XP de forma que os roteiros de teste possam ser modificados e rodados novamente para cada uma das frequentes iterações de desenvolvimento.
- Ocasionalmente, uma organização pode não estar interessada em ferramentas de teste *software* comerciais. Com o propósito de gerenciar a garantia da qualidade, o desenvolvimento de uma ferramenta de teste automatizada melhora significativamente o projeto de desenvolvimento.

O modelo do ciclo de vida do desenvolvimento de *software* também afeta a extensão de um processo de teste automatizado, já que (LI; WU, 2004, p. 3-4):

- Geralmente, modificações ocorrem com frequência no início de um projeto. O objetivo é utilizar uma ferramenta que gere os roteiros de teste automaticamente, que refletiria as mudanças e significativamente melhoraria a eficiência do teste com produtos não estáveis.
- As ferramentas de teste disponíveis comercialmente se dizem capazes de testar componentes da interface gráfica com o usuário (*graphical user interface* - GUI). Na maior parte do tempo, os usuários precisam gravar uma série de cliques do mouse e teclas digitadas. Os dados de teste dos roteiros gravados são geralmente estáticos. Às vezes, o roteiro de teste gravado precisa ser revisado e corrigido antes que ele possa realizar o teste. Desta forma, o uso e reuso destas ferramentas em um produto não estável pode ser um pesadelo. Uma ferramenta de testes automatizados deve ser capaz de reconhecer e verificar os componentes GUI por si própria e gerar um roteiro de testes orientado a dados (*data driven*).
- A medida que as tecnologias avançam, a programação se torna mais fácil. O ciclo de vida do desenvolvimento se torna mais curto, dando aos testadores menos tempo para testar seus produtos. Uma ferramenta de teste automatizado deve aliviar um pouco desta pressão de forma que os testadores tenham mais tempo para identificar as áreas de maior risco do projeto.

Fewster e Graham (1999) listam uma série de benefícios que podem ser trazidos pela automação dos testes, além de possibilitar que algumas tarefas ligadas ao teste sejam performadas mais eficientemente do que manualmente:

- A realização de testes de regressão, ou seja, rodar testes existentes em uma nova versão do programa, com um mínimo de esforço, visto que os testes já existem prontos para versões anteriores do *software*.
- Executar mais testes com maior frequência. Um benefício da automação é a habilidade de rodar mais testes em menos tempo e portanto tornar possível rodá-los mais frequentemente. Isto trará uma maior confiança no sistema. Muitas pessoas assumem

que elas irão executar os mesmos testes mais rápidos com a automação. De fato, elas tendem a executar mais testes, e estes são executados mais frequentemente.

- Executar testes que serão difíceis ou impossível de executar manualmente. Por exemplo, executar um teste ao vivo de um sistema on-line com 200 usuários simultaneos pode ser impossível, mas a simulação destes 200 usuários pode ser realizada utilizando um teste automatizado.
- Melhor uso dos recursos. A automação de tarefas enfadonhas, tais como entradas repetidas das mesmas entradas, trazem uma maior acurácia, bem como aumentam a moral das pessoas, bem como libera testadores qualificados para colocar mais esforço em planejar melhor os casos de teste a serem executados. Sempre haverão alguns testes que são melhores de serem executados manualmente. Nestes casos os testadores poderão fazer uma melhor trabalho no teste manual se houver muito menos testes para serem executados desta forma.
- Consistência e repetibilidade dos testes. Os testes que são repetidos automaticamente serão repetidos exatamente cada uma das vezes (pelo menos as entradas serão; as saídas podem ser diferentes ao longo do tempo, por exemplo). Isto dá um nível de consistência aos testes que é muito difícil de se obter manualmente. Os mesmos testes podem ser executados em diferentes configurações de hardware, utilizando diferentes sistemas operacionais, ou diferentes bancos de dados. Isto dá uma consistência de qualidade para os produtos multi-plataforma que é virtualmente impossível de atingir com os testes manuais. A imposição de um bom regime de teste automatizado pode também garantir normas consistentes tanto para o teste como para o desenvolvimento. Por exemplo, a ferramenta pode verificar se o mesmo tipo de funcionalidade foi implantada da mesma forma em cada aplicação ou programa.
- Reutilização dos testes. O esforço colocado em decidir o que testar, planejar os testes, e construí-los pode ser distribuido através de muitas execuções destes testes. É importante investir um tempo na verificação da confiabilidade dos testes que serão reutilizados. Isto também é importante nos testes manuais, mas um teste automatizado será reutilizado muito mais vezes do que o mesmo teste repetido manualmente.
- Antecipação da entrada no mercado (*time to market*). Uma vez que um conjunto de testes tenha sido automatizado, ele pode ser repetido muito mais rapidamente do que se fosse executado manualmente, de forma que o tempo gasto no teste pode ser encurtado (sujeito a outros fatores como a disponibilidade dos desenvolvedores para corrigir os defeitos).

- Aumento da confiança. Conhecendo-se que um conjunto extensivo de testes automatizados foi executado com sucesso, pode haver uma maior confiança que não ocorrerão surpresas desagradáveis quando o sistema for lançado (considerando que os testes rodados sejam bons). Em resumo, testes mais completos podem ser alcançados com menos esforço, trazendo aumentos tanto na qualidade quanto na produtividade.

Por outro lado, existe uma série de problemas que podem ser encontrados na tentativa de automatizar o teste, segundo Fewster e Graham (1999). Os autores comenta uma série de problemas mais comuns:

- Expectativas não realistas. Existe uma tendência no excesso do otimismo sobre os resultados na compra de uma nova ferramenta. Se as expectativas não forem realistas, não importará o quão bem uma ferramenta seja implantada e utilizada do ponto de vista técnico que elas não serão satisfeitas.
- Prática pobre de teste. Se a prática de teste for pobre, com testes mal organizados, pouca documentação ou com a mesma inconsistente, e testes que não são muito bons em encontrar defeitos, a automação de testes não será uma boa idéia. É de longe melhor aumentar primeiro a efetividade do teste ao invés de aumentar a eficiência de um teste pobre. O autor cita que a automação do caos apenas resulta em um caos mais rápido.
- Expectativa que os testes automatizados irão encontrar vários novos defeitos. Um teste possui as maiores chances de encontrar um defeito em sua primeira execução. Se um teste já foi executado e foi bem sucedido, rodar o mesmo teste novamente possui uma probabilidade menor de encontrar um novo defeito, ao menos que o testes esteja exercitando um código que tenha sido alterado ou que possa ser afetado por uma mudança feita em uma parte diferente do *software*, ou ainda, esteja sendo executado em um ambiente diferente. As ferramentas de execução são ferramentas de *replay*, ou seja, ferramentas de testes de regressão. Seu uso é mais voltado para testes de repetição que já tenham sido executados. Isto é uma coisa muito importante para fazer, mas não aumenta as chances de encontrar um grande número de novos defeitos, particularmente quando a execução for no mesmo ambiente de *software* e *hardware* que na execução anterior. Por outro lado, o autor reforça que os testes que não encontram defeitos têm o seu valor, embora um teste bem planejado deve ser direcionado em tentar encontrar os defeitos. Saber que um conjunto de testes foi bem sucedido novamente dá confiança que

o *software* está ainda funcionando tão bem quanto estava antes, e que as mudanças não tiveram efeitos não previstos.

- Falsa sensação de segurança. O fato da *suite* de teste ter sido executada sem encontrar nenhum defeito, não significa que não existem defeitos no *software*. Os testes podem ser incompletos, ou podem conter defeitos neles mesmos. Se os resultados esperados forem incorretos, os testes automatizados simplesmente preservarão aqueles resultados defeituosos indefinidamente.
- Manutenção dos testes automatizados. Quando o *software* é mudado, frequentemente é necessário atualizar alguns, ou até mesmo todos, os testes de forma que eles possam ser re-executados de maneira bem sucedida. Isto é particularmente verdade para os testes automatizados. O esforço de manutenção do teste tem sido a morte de muitas iniciativas de automação de testes. Quando é necessário mais esforço para atualizar os testes do que o necessário para re-executar aqueles testes manualmente, a automação do teste é abandonada.
- Problemas técnicos. As ferramentas de execução de testes comerciais são *softwares*. Como tal elas não são imunes à defeitos ou problemas de suporte. Talvez seja duplamente desapontante descobrir que uma ferramenta de teste não tenha sido tão bem testada, mas infelizmente, isto ocorre. Além de problemas técnicos com as ferramentas de teste propriamente dito, pode ocorrer problemas técnicos com o *software* que é alvo do teste. Se o *software* não foi desenhado e construído com testabilidade em mente, pode ser muito difícil testá-lo, tanto manualmente como automaticamente. Tentar usar ferramentas para testar tais *softwares* pode adicionar complicações que podem tornar a automação de teste ainda mais difícil.
- Assuntos organizacionais. A automação do teste não é um exercício trivial, e ela precisa ser bem suportada pela administração e implementada na cultura da organização. Um tempo precisa ser alocado para a escolha das ferramentas, para o treinamento, para experimentar e aprender o que funciona melhor, e para promover a utilização da ferramenta na organização. Um esforço de automação provavelmente não terá sucesso ao menos que exista uma pessoa que é o ponto focal para a utilização da ferramenta. Tipicamente, esta é uma pessoa que está entusiasmada com o teste automatizado, e que irá comunicar seu entusiasmo dentro da empresa. Esta pessoa pode estar envolvida em selecionar qual ferramenta comprar, e será ativa na promoção de seu uso interno. A automação do teste é um assunto de infra-estrutura, não apenas um assunto de projeto. Em organizações grandes, a automação de teste raramente pode ser justificada com base

em um simples projeto, pois o projeto suportará todos os custos dos problemas de sua implantação e se aproveitará pouco dos benefícios. Se o escopo da automação de teste é apenas para um projeto, as pessoas depois serão alocadas em novos projeto, então o ímpeto será perdido. A automação de teste frequentemente falha justamente no momento que ela devia prover o maior valor, ou seja, quando o *software* é atualizado. Normas são necessárias para garantir maneiras consistentes de utilizar as ferramentas na organização. De outra forma, cada grupo pode desenvolver abordagens diferentes para a automação dos testes, tornando difícil transferir ou compartilhar os testes automatizados ou os testadores entre os grupos. Até mesmo um pequeno problema administrativo como possuir poucas licenças para as pessoas que desejam utilizar a ferramenta pode ser muito impactante para o sucesso e para o custo do esforço da automação de teste. As percepções do esforço de trabalho podem também mudar. Se um teste é executado durante a noite, então quando os testadores chegarem pela manhã, eles precisarão gastar mais tempo olhando para os resultados do teste. Este tempo de análise do teste é agora claramente visível como uma atividade separada. Quando esses testes eram rodados manualmente, este tempo de análise do teste estava embutido na atividade de execução do testes, e não era visível. Sempre que uma nova ferramenta (ou até mesmo um novo processo) for implementado, inevitavelmente existirão ajustes que precisam ser feitos para adaptar às novas maneiras de trabalhar, os quais precisam ser gerenciados.

Li e Wu (2004) apresenta a seguintes deficiências geralmente presentes:

- Os roteiros de teste necessitam de correções;
- Nenhuma das ferramentas pode realizar um teste completo independentemente;
- As ferramentas utilizam processos que podem não ser consitentes com o modelo de software da organização;
- O processo de engenharia reversa é separado da geração de roteiro de testes.

Dustin (2003) comenta que apesar das ferramentas de testes funcionais serem as mais conhecidas e comentadas (também conhecidas como ferramenta de captura e *playback*), existe uma série de outros tipos de ferramentas para suportar o ciclo de vida de teste que são apresentadas na Tabela 24.

Tabela 24 - Ferramentas de Teste

Tipo de ferramenta	Descrição
Geradores de procedimentos de teste	Gera procedimentos de teste a partir de requisitos e modelos.
Analizadores de cobertura de código e intrumentadores de código	Identificam código não testados e suportam os testes dinâmicos.
Detecção de vazamentos de memória (Memory-Leak Detection)	Verifica se a aplicação está gerenciando adequadamento os recursos de memória.
Ferramentas de geração de relatório de métricas	Lê o código fonte e mostra informações de métricas, como complexidade do fluxo de dados, estrutura de dados, e fluxo de controle. Pode prover métricas de tamanho de código em termos de número de módulos, operandos, operadores e linhas de código.
Ferramentas de medição de usabilidade	Perfil de uso, análise de tarefas, prototipação e walk-throughs de usuário.
Geradores de dados de teste	Geração de dados de teste.
Ferramentas de gerenciamento de testes	Provê funcionalidades de gerenciamento dos testes tais como documentação dos procedimentos de teste, armazenamento e rastreabilidade.
Ferramentas de teste de rede	Monitoração, medição, teste e diagnóstico de performance de rede de dados.
Ferramentas de teste de gravação e Playback de interação com a interface gráfica com o usuário (GUI)	Automatiza os testes gravando as interações do usuário com a intergace gráfica do software, permitindo a reprodução automática.
Ferramentas de teste de carga, performance, e stress	Realização de teste de carga, performance, e stress.
Ferramentas especializadas	Ferramentas de arquiteturas específicas que provêm testes especializados de arquiteturas ou tecnologias específicas, tal como sistemas embarcados.

FONTE: DUSTIN, 2003

Tsao *et al* (2003, p. 157-186) também apresenta uma classificação dos tipos de ferramentas de suporte e automação dos testes. A Tabela 25 apresenta esta classificação.

Tabela 25 – Uma classificação das ferramentas de automação de testes de *Software*

Tipo de ferramentas de teste	Descrições das ferramentas
Gerenciamento da informação de teste	Suporte aos engenheiros de teste e profissionais de garantia da qualidade para criar, atualizar, e manter diversas informações de teste, incluindo casos de teste, roteiros de teste, dados de teste, resultados de teste, e problemas descobertos.

Controle e execução de teste	Ferramentas para auxiliar os engenheiros a configurar e a executar os testes, bem como coletar e validar os resultados.
Geração de teste	Ferramentas para gerar testes de forma automática.
Análise de cobertura de teste	Ferramentas para análise da cobertura do teste durante o processo de teste baseado em critérios de teste selecionados.
Medição e teste de performance	Ferramentas para realizar testes e medidas da performance de *softwares*.
Simuladores	Programas desenvolvidos para simular as funções e comportamento de sistemas externos, ou sub-sistemas/componentes dependentes para o teste de um software.
Testes de regressão	Ferramentas de teste para automação das atividades de teste de regressão, incluindo a gravação do teste e sua reprodução.

FONTE: TSAO *et al*, 2003, p. 166

Já na Tabela 26, Tsao *et al* (2003, p. 167) teve a preocupação de elencar os diferentes *vendors* de cada tipo de ferramenta de teste, bem como as ferramentas disponíveis no mercado.

Tabela 26 – Diferentes tipos de ferramentas de testes e *vendors*

Tipos de ferramentas de teste	***Vendors***	**Ferramentas de teste**
Ferramentas de gerenciamento de problemas	Rational Inc.	ClearQust, ClearDDTS
	Microsoft Corp.	PVCS Tracker
	Imbus AG	Imbus Fehlerdatenbank
Ferramentas de gerenciamento de testes	Rational Inc.	TestManager
	Mercury Interactive	TestDirectory
Ferramentas de gerenciamento de suites de teste	Evalid	TestSuiter
	Rational Inc.	TestFactory
	SUN	JavaTest, JavaHarness
Ferramentas de teste de caixa branca	McCabe & Associates	McCabe IQ2
	IBM	IBM COBOL Unit Tester
		IBM ATC
		Coverage Assistant
		Source Audit Assistant
		Distillation Assistant
		Unit Test Assistant
Ferramentas de execução de testes	OC Systems	Aprob
	Softbridge	ATF/TestWright
	AutoTester	AutoTester
	Rational Inc.	Visual Test

	SQA	Robot
	Mercury Interactive	WinRunner
	Sterling *Software*	Vision TestPro
	Compuware	QARun
	Seque *Software*	SilkTest
	RSW *Software* Inc.	e-Test
	Cyrano Gmbh	Cyrano Robot
Ferramentas de análise de cobertura de testes	Case Consult Corp.	Analyzer, Analyzer Java
	OC Systems	Aprob
	IPL *Software* Product Group	Cantata/Cantata++
	ATTOL Testware SA	Coverage
	Compuware NuMega	TruCoverage
	Software Research	TestWorks Coverage
	Rational Inc	PureCoverage
	SUN	JavaScope
	ParaSoft	TCA
	Software Automation Inc.	Panorama
Ferramentas de testes de carga e performance	Rational Inc.	Rational Suite PerformanceStudio
	InterNetwork AG	sma@rtTest
	Compuware	QA-Load
	Mercury Interactive	LoadRunner
	RSW *Software* Inc.	e-Load
	SUN	JavaLoad
	Seque *Software*	SilkPerformer
	Client/Server Solutions, Inc.	Benchmark Factory
Ferramentas de teste de regressão - GUI *record/replay*	IBM	Regression Testing Tool(ARTT)
		Distillation Assistant
	Software Research	eValid
	Mercury Interactive	Xrunner
	Astra	Astra QuickTest
	AutoTester	AutoTester, AutoTester One

FONTE: TSAO *et al*, 2003, p. 167-168

Fewster e Graham (1999) mencionam que ferramentas de suporte ao teste estão disponíveis para cada estágio do ciclo de vida de desenvolvimento de *software*. Para ilustrar a colocação, eles apresentam uma diagrama com os diferentes estágios de desenvolvimento de *software* e os diferentes tipos de ferramenta aplicáveis a cada um destes (Figura 32).

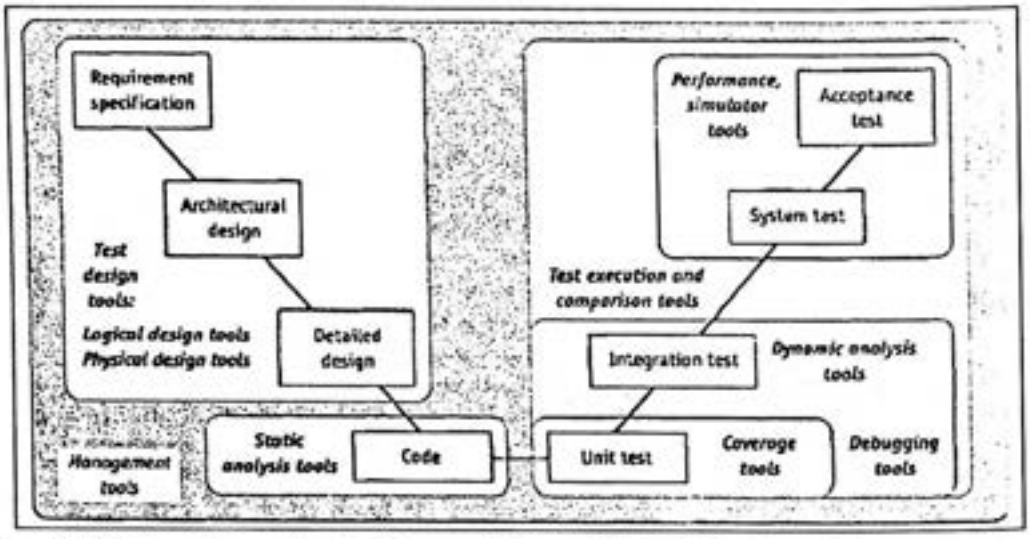

Figura 32 - Onde as ferramentas de teste se encaixam no ciclo de vida de desenvolvimento de *software*
FONTE: FEWSTER e GRAHAM, 1999

As ferramentas de modelagem do teste (*test design tools*) ajudam a derivar as entradas de teste ou dados de teste. As ferramentas de modelagem lógica (*logical design tools*) trabalham a partir da lógica de uma especificação, uma interface ou código, e são algumas vezes conhecidas como geradores de casos de teste (*test case generators*). As ferramentas de modelagem física (*physical design tools*) manipulam os dados existentes ou geram dados de teste. Por exemplo, uma ferramenta que pode extrair registros aleatórios de um banco de dados seria uma ferramenta de modelagem física. Uma ferramenta que pode derivar as entradas de teste das especificações seria uma ferramenta de modelagem lógica (FEWSTER; GRAHAM, 1999).

Dentre as ferramentas de gerenciamento de teste estão incluídas aquelas para assistir o planejamento dos testes, manter controle dos testes que estão sendo executados, e assim por diante. Esta categoria de ferramentas, segundo o autor, também inclui ferramentas para cuidar da rastreabilidade dos testes aos requisitos, modelagem (*design*), código, bem como ferramentas de rastreamento de defeitos (FEWSTER; GRAHAM, 1999).

As ferramentas de análise estática de código sem executá-lo detectam certos tipos de defeitos de maneira mais eficiente e econômica que de outras formas. Tais ferramentas também calculam várias métricas para o código em questão, como a complexidade ciclomática de McCabe, métricas de Halstead, métricas de duplicidade e muitas outras (FEWSTER; GRAHAM, 1999)

As ferramentas de cobertura (*coverage tools*) avaliam quanto do *software* em teste foi exercitado pelos conjuntos de testes. As ferramentas de cobertura são normalmente usadas no nível do teste unitário (FEWSTER; GRAHAM, 1999).

As ferramentas de *debug*, segundo o autor, não são ferramentas de teste, pois, em sua visão, o *debug* não faz parte da atividade de teste, já que os testes identificam os defeitos e o *debug* os remove, sem contar que é uma atividade de desenvolvimento e não de teste. No entanto, as ferramentas de *debug* são frequentemente utilizadas no teste, especialmente na tentativa de isolar defeitos em níveis muito baixos. As ferramentas de *debug* permitem que o desenvolvedor realize um acompanhamento passo a passo no código do *software* executando uma instrução de cada vez e olhando os conteúdos das variáveis e memória (FEWSTER; GRAHAM, 1999).

As ferramentas de análises dinâmicas avaliam o sistema enquanto o *software* está sendo executado. Por exemplo, ferramentas que podem detectar vazamentos de memória (*memory leaks*) são ferramentas de análises dinâmicas. Um vazamento de memória ocorre se um programa não libera blocos de memória quando ele deveria, de forma que estes "vazam" do *pool* de blocos de memória disponível para todos os programas. Eventualmente, o programa pode acabar ocupando toda a memória do computador, de forma que nenhum outro programa possa rodar, levando o sistema como um todo a falhas e eventualmente à indisponibilidade e à reinicialização do computador (FEWSTER; GRAHAM, 1999).

Os simuladores são ferramentas que fazem com que seja possível testar partes de um sistema, o que não seria possível no mundo real. Por exemplo, os procedimentos de fusão de uma planta de uma usina nuclear podem ser testadas em um simulador (FEWSTER; GRAHAM, 1999).

Uma outra classe de ferramentas tem a ver com o que se pode chamar de teste de capacidade (*capacity testing*). As ferramentas de teste de performance (*performance testing tools*) medem o tempo de duração de vários eventos. Por exemplo, elas podem mensurar tempos sob uma condição típica ou de carga. As ferramentas de teste de carga (*load testing tools*) geram tráfego no sistema. Por exemplo, elas podem gerar um número de transações que representam os níveis típicos ou máximo. Este tipo de ferramenta pode ser utilizada para testes de volume ou de estresse (*volume and stress testing*) (FEWSTER; GRAHAM, 1999).

As ferramentas de execução e comparação de testes (*test execution and comparison tools*) fazem com que os testes possam ser executados automaticamente e as saídas dos testes possam ser comparadas às saídas esperadas. Estas ferramentas são aplicáveis à execução dos testes em qualquer nível: unitário, integração, sistema, ou aceite. As ferramentas de captura e *replay* (*capture and replay tools*) são ferramentas de execução e comparação de testes. Estas são os tipos de ferramenta de teste mais popular em uso (FEWSTER; GRAHAM, 1999).

2.6.11 Processos de teste

Diversos são os processos de teste encontrados na literatura. Grande parte deles se apóia em 4 atividades básicas: planejamento, modelagem, execução e análise/reportamento (DE YOUNG; RÄTZMANN, 2003; HORCH, 2003, p. 3; CRAIG; JASKIEL, 2002).

A Figura 33 mostra uma representação esquemática e simplificada das atividades relacionadas ao teste no contexto do desenvolvimento de *software* apresentada em (HORCH, 2003, p. 3).

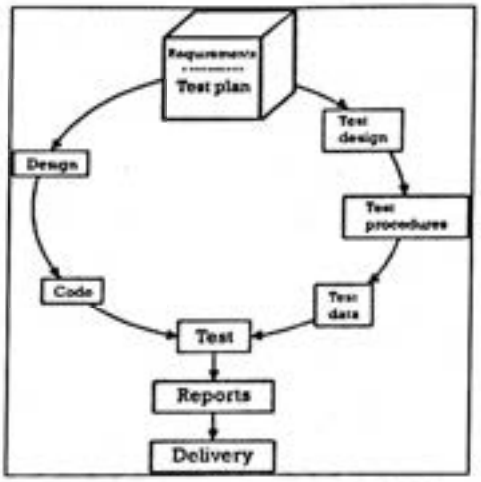

Figura 33 - Processo de teste simplificado
FONTE: HORCH, 2003, p. 3

Horch (2003, p. 3-4) descreve que as atividades de planejamento do teste (*test planing*) se iniciam durante a fase de requisito e caminham em paralelo ao desenvolvimento dos requisitos. A cada requisito gerado, o método correspondente de teste para ele deve ser considerado. Um requisito é falho se não é testável, segundo o autor.

Já a modelagem dos testes (*test design*) se inicia juntamente com a moldelagem do *software* (*software design*), de forma que a medida que o modelo do *software* vai tomando forma, os casos de teste, cenários e dados vão sendo desenvolvidos para exercitarem o *software* sendo modelado (HORCH, 2003, p. 3-4).

Horch (2003, p. 4) menciona que os testes, na verdade, se iniciam com o *debug*, bem como com os testes unitários e de módulo realizados pelo programador, geralmente de uma maneira informal. A execução formal do teste geralmente se inicia com os testes integrados, segundo o autor.

Por fim, um teste de aceite é realizado, concentrando-se geralmente na funcionalidade e usabilidade real do sistema no ambiente do usuário (HORCH, 2003, p. 4).

Fewster e Graham (1999) apresentam as principais atividades associadas ao teste de *software* em um diagrama (Figura 34) e afirmam que a forma como estas atividades são conduzidas varia de acordo com a organização. Em alguns casos elas são realizadas formalmente, já em

outros, elas são informais ao ponto de eventualmente serem quase caóticas. De qualquer forma, como os autores dizem, elas são geralmente mais ou menos da forma descrita.

Craig e Jaskiel (2002) apresentam uma metodologia denominada STEP (Systematic Test and Evaluation Process), no qual sua obra é apoiada, criada em 1985 a partir da norma IEEE Std. 829-1983 (*Standard for Software Test Documentation*) e autalizada em cima de versões posteriores (IEEE Std. 828-1998 e IEEE Std. 1008-1987 *Standard for Software Unit Testing*). Seu foco, segundo descrevem Craig e Jaskiel (2002), é estressar o potencial de prevenção do teste, tendo como objetivos secundários a detecção de defeitos e a demonstração de capacidade.

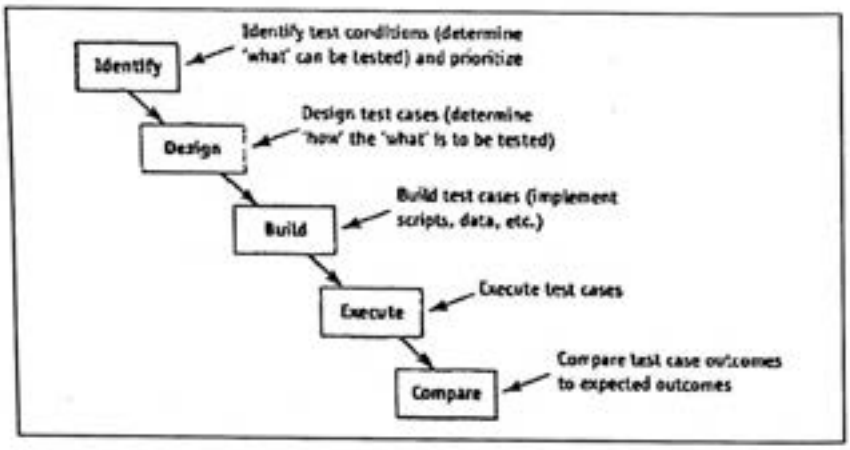

Figura 34 - Processo de teste com atividades considerado como um ciclo de vida de desenvolvimento de casos de teste

FONTE: FEWSTER; GRAHAM, 1999

De acordo com os autores, as principais diferenças entre o STEP e as práticas de teste comuns são apontadas na Tabela 27 (CRAIG; JASKIEL, 2002).

Tabela 27 – Principais diferenças entre STEP e a prática na indústria

Metodologia	Foco	Momento do planejamento	Momento da aquisição	Cobertura	Visibilidade
STEP	Prevenção e gerenciamento de risco	Começa durante a definição do requisito	Começa durante a definição do requisito	Conhecida (Relativa aos inventários)	Totalmente documentada e avaliada
Práticas na indústria	Demonstração e detecção	Começa depois do término da modelagem do software	Começa depois do término da modelagem ou da codificação	Amplamente desconhecida	Amplamente não documentada e com pouca ou nenhuma avaliação

FONTE: CRAIG; JASKIEL, 2002

Para Craig e Jaskiel (2002) estas diferenças são muito significativas e difíceis de serem colocadas em prática, no entanto, os benefícios comprovados pelo STEP são igualmente significativos e fazem jus a toda a dificuldade e investimento.

Outro processo de teste utilizado atualmente é o descrito pelo Rational Unified Process (RUP) (RATIONAL, 2002). O fluxo de atividades para os testes contemplado por esta metodologia de desenvolvimento de *software* iterativa e orientada a risco é mostrado na Figura 35.

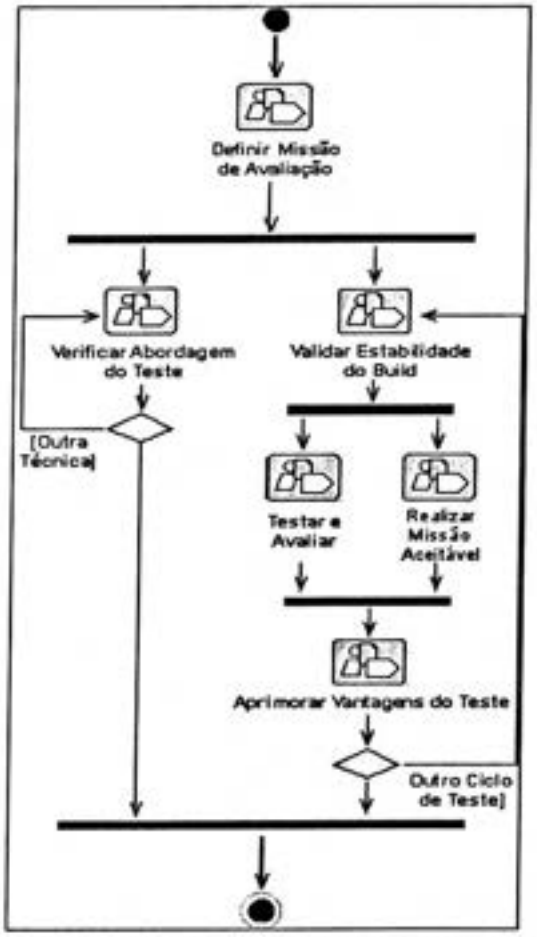

Figura 35 – Fluxo de atividades de Testes do RUP

FONTE: RATIONAL, 2002

Além de definir detalhadamente cada uma atividades dentro do fluxo de trabalho, o RUP (RATIONAL, 2002) também especifica os papéis (Figura 36) de cada profissional envolvido nas atividades de teste de *software*, bem como as atividades de responsabilidade de cada um

destes papéis (Figura 36), e ainda, os artefatos ou documentos (Figura 37), utilizados e gerados em cada uma das etapas do processo de teste (RATIONAL, 2002).

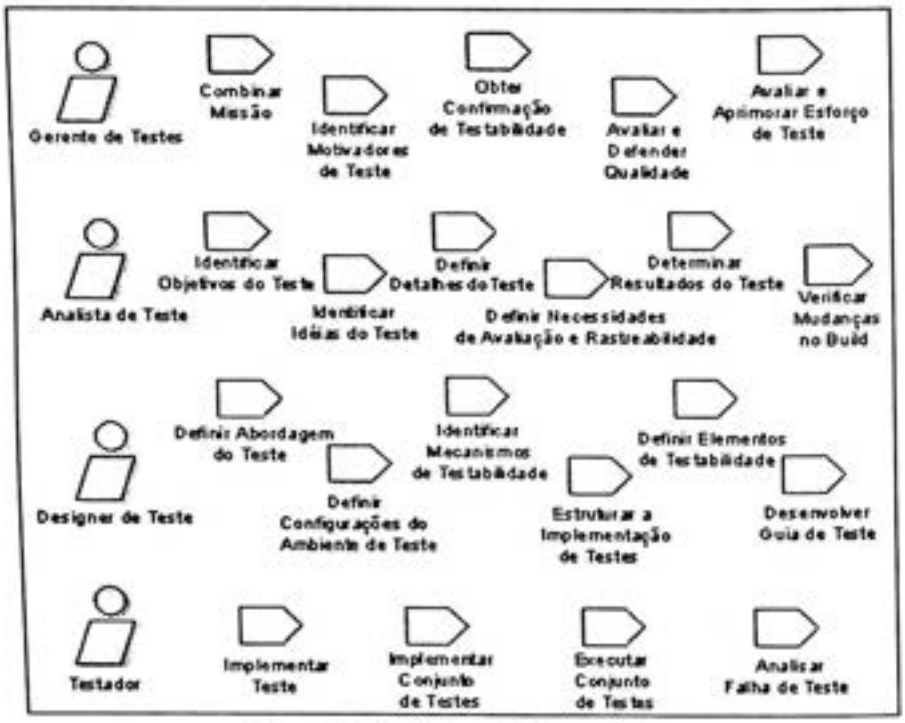

Figura 36 - Papéis envolvidos nos Testes

FONTE: RATIONAL, 2002

A finalidade do detalhamento do fluxo de trabalho denominado "definir a missão de avaliação" é identificar o foco adequado do esforço de teste para a iteração e estabelecer consenso com os envolvidos sobre as metas correspondentes que conduzirão o esforço de teste (KRUCHTEN, 1999).

Para cada iteração, o foco principal do trabalho será (KRUCHTEN, 1999):

- Identificar os objetivos, e os produtos liberados, do esforço de testes
- Identificar uma boa estratégia de utilização de recursos
- Definir o escopo e o limite adequados para o esforço de teste
- Descrever o método que será usado
- Definir como o progresso será monitorado e avaliado

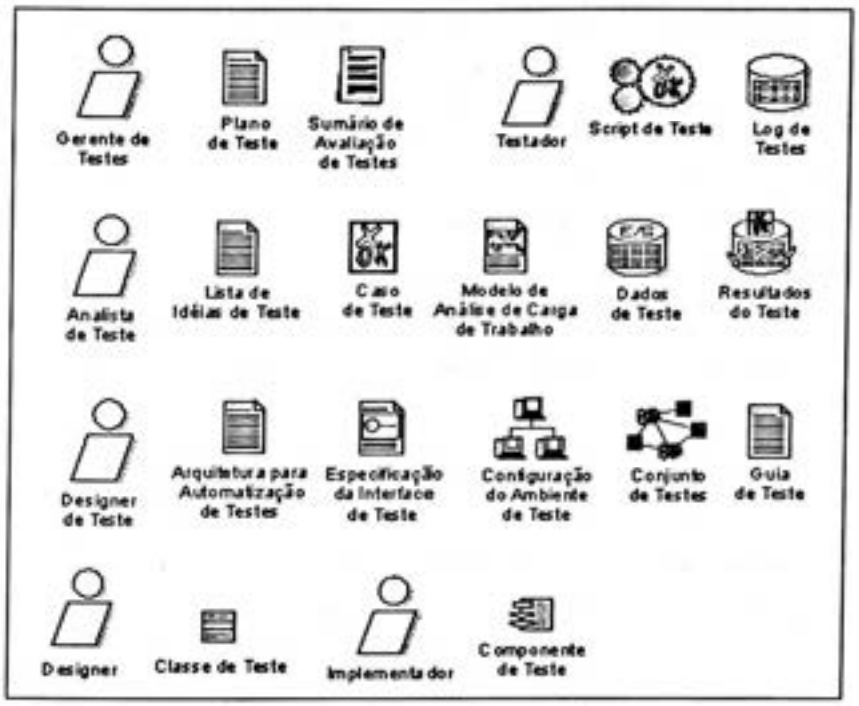

Figura 37 - Artefatos de Testes do RUP

FONTE: RATIONAL, 2002

A finalidade do detalhamento do fluxo de trabalho denominado "verificar a abordagem do teste" é demonstrar que as várias técnicas descritas na Abordagem do Teste facilitarão o teste exigido. Ele demonstra que a abordagem funcionará, produz resultados precisos e é adequado para os recursos disponíveis. O objetivo é compreender as restrições e limitações de cada técnica e encontrar uma solução de implementação adequada ou encontrar técnicas alternativas que possam ser implementadas. Isso ajuda a diminuir o risco de descobrir muito tarde no ciclo de vida do projeto que a abordagem do teste não funciona (KRUCHTEN, 1999).

Para cada iteração, o foco principal do trabalho será (KRUCHTEN, 1999):

- Verificar o quanto antes que a Abordagem de Teste pretendida funcionará e produzirá resultados valiosos
- Estabelecer a infra-estrutura básica para possibilitar e dar suporte à Abordagem do Teste

- Obter comprometimento da equipe de desenvolvimento para fornecer e dar suporte à testabilidade exigida para atingir a Abordagem de Teste
- Identificar escopo, fronteiras, limitações e restrições de cada técnica

A finalidade do detalhamento do fluxo de trabalho "validar estabilidade do *build*" é validar *build* tornando-o estável o suficiente para iniciar teste detalhado e o esforço de avaliação. Esse trabalho também é chamado de teste de regressão, teste de verificação do *build*, teste de regressão do *build*, verificação de limpeza ou aceitação no teste. Esse trabalho ajuda a economizar recursos de teste em esforço inútil (KRUCHTEN, 1999).

Para cada *build* a ser testado, o foco do trabalho será (KRUCHTEN, 1999):

- Fazer uma avaliação da estabilidade e testabilidade do *build*
- Formar uma compreensão inicial — ou confirmar a expectativa — do trabalho de desenvolvimento feito no *build*
- Tomar uma decisão para aceitar o *build* como adequado para uso — com base na missão de avaliação — nos testes seguintes ou realizar mais testes em um *build* anterior

A finalidade do detalhamento do fluxo de trabalho "testar e avaliar" é atingir a amplitude e o detalhamento apropriados do esforço de teste para permitir uma avaliação suficiente do item de teste-alvo - em que a avaliação suficiente é controlada pelos motivadores de teste e pela missão de avaliação. Normalmente realizado uma vez por ciclo de teste, esse trabalho envolve a execução do trabalho tático central do esforço de teste e avaliação, ou seja, a implementação, a execução e a avaliação de testes específicos e o relatório correspondente dos incidentes encontrados (KRUCHTEN, 1999).

Para cada ciclo de teste, o foco principal do trabalho será (KRUCHTEN, 1999):

- Fornecer avaliação contínua dos itens de teste-alvo
- Registrar as informações necessárias para diagnosticar e resolver os problemas identificados
- Atingir a amplitude e o detalhamento apropriados no trabalho de teste e avaliação
- Fornecer feedback nas áreas de risco potencial para a qualidade

A finalidade do detalhamento de fluxo de trabalho "realizar missão aceitável" é oferecer um resultado útil de avaliação para os envolvidos no esforço de teste. Esse resultado é examinado de acordo com a missão de avaliação. Na maioria dos casos, isso significará concentrar seus esforços para ajudar a equipe do projeto a atingir os objetivos do plano de iteração aplicável ao ciclo de teste atual.

Para cada ciclo de teste, o foco principal do trabalho será (KRUCHTEN, 1999):

- Priorizar ativamente o conjunto mínimo de testes necessários que deve ser aplicado para atingir a missão de avaliação
- Defender a resolução de problemas importantes que têm um grande impacto negativo na missão de avaliação
- Defender a qualidade adequada
- Identificar regressões de qualidade introduzidas entre os ciclos de testes
- Onde for pertinente, rever a missão de avaliação considerando a avaliação a fim de fornecer informações importantes para a equipe do projeto

A finalidade deste detalhamento do fluxo de trabalho "aprimorar vantagens do teste" é manter e melhorar as vantagens do teste. Isso será particularmente importante caso a intenção seja reutilizar as vantagens desenvolvidas no ciclo atual de teste em ciclos subseqüentes (KRUCHTEN, 1999).

Para cada ciclo de teste, o foco principal do trabalho será (KRUCHTEN, 1999):

- Adicionar o conjunto mínimo de testes adicionais para validar a estabilidade dos *builds* seguintes
- Montar roteiros de teste em outros conjuntos de testes apropriados
- Remover as vantagens do teste que não tenham mais finalidade ou cuja manutenção se tenha tornado dispendiosa
- Manter Configurações de ambiente de teste e conjuntos de dados de teste
- Explorar as oportunidades para reutilização e melhorias na produtividade
- Realizar manutenção geral das vantagens de automatização de testes e fazer melhorias nelas

- Documentar as lições aprendidas — tanto as práticas boas quanto as ruins descobertas durante o ciclo de teste.

2.6.12 Técnicas de teste

Muitos métodos de teste que foram desenvolvidos para o teste de hardware podem ser adaptados para o teste de software de acordo com Beizer (1990, p. 309).

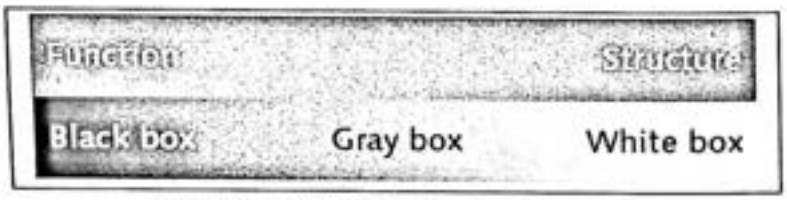

Figura 38 - Classificação das técnicas de teste
FONTE: DE YOUNG; RÄTZMANN, 2003; p. 49

Diversas obras classificam os testes em dois tipos, os testes de caixa preta e os testes de caixa branca, onde o primeiro se baseia apenas nos requisitos e especificações, sem a necessidade de conhecer os caminhos internos, estrutura, ou a implementação do *software* em teste, enquanto o segundo se baseia nos caminhos internos, estruturas e na implementação do *software* em teste, necessitando muitas vezes de habilidades de programação. (DE YOUNG; RÄTZMANN, 2003, p. 49; COPELAND, 2004)

Alguns autores apresentam ainda um tipo adicional de teste, o chamado teste de caixa cinza (*gray box testing*). (DE YOUNG; RÄTZMANN, 2003; COPELAND, 2004)

Hung Q. Nguyen (Nguyen, 2001 apud DE YOUNG; RÄTZMANN, 2003, p. 50) define o teste de caixa cinza como métodos e ferramentas derivados do conhecimento interno de aplicações e ambientes com os quais elas interagem, nas quais pode-se aplicar os testes de caixa preta para aumentar a produtividade em encontrar os erros e analisar os *bugs*.

Nesta abordagem, segundo Copeland (2004), deve-se olhar dentro da "caixa" em teste apenas o suficiente para entender como ela foi implementada e utilizar este conhecimento para escolher os testes de caixa preta mais efetivos.

Em algumas obras (DE YOUNG; RÄTZMANN, 2003, p. 50; BEIZER, 1990, p. 11), os testes de caixa branca também são conhecidos como testes estruturais. De Young e Rätzmann (2003, p. 50) chamam os testes de caixa preta também de testes funcionais. Já Hutcheson (2003, p. 135) também denomina os testes de caixa preta como testes comportamentais (*behavioral testing*).

Copeland (2004) menciona que os testes de caixa preta podem ser aplicados em todos os níveis do desenvolvimento de *software*: unitário, integração, sistema e aceite.

O processo geral de teste de caixa preta, apresentado por Copeland (2004), pode ser sintetizado nas seguintes atividades:

- análise de requisitos e especificações;
- escolha de entradas válidas baseadas na especificação para determinar que o sistema em teste as processe corretamente. Entradas inválidas devem também ser escolhidas para verificar se o sistema em teste as detecta e as trata da forma devida;
- as saídas esperadas para as entradas são determinadas;
- os testes são construídos para as entradas selecionadas;
- os testes são executados;
- as saídas produzidas são comparadas com as esperadas;
- uma determinação é feita a respeito do funcionamento adequado do sistema em teste.

Uma das desvantagens apresentadas por Copeland (2004) na utilização das técnicas de teste de caixa preta é que o testador nunca pode saber quanto do sistema foi testado. Além disto, Copeland (2004) menciona que para encontrar "todos" os defeitos utilizando esta técnica, o testador deve criar todas as possíveis combinações de entrada, tanto válidas como inválidas, o que é praticamente impossível. Por outro lado, ele ressalta a vantagem que as técnicas formais de teste de caixa preta direcionam o testador a escolher subconjuntos de teste que são tanto eficazes como eficientes em encontrar defeitos, principalmente se comparando com a escolha aleatória de testes, ajudando a retornar o investimento em teste.

Já o processo geral de teste utilizando uma técnica de caixa branca é apresentado por Copeland (2004) como uma seqüência de atividades que pode ser sintetizada por (COPELAND, 2004):

- a implementação do *software* é analisada;
- os caminhos através do *software* são identificados;
- as entradas são escolhidas de forma que façam com que o *software* execute os caminhos selecionados. Isto é chamado de sensibilização do caminho (*path sensitization*). Os resultados esperados para estas entradas são determinados;
- os testes são executados;
- as saídas produzidas são comparadas com as esperadas;
- uma determinação é feita a respeito do funcionamento adequado do sistema em teste.

Os testes de caixa branca podem ser aplicados em todos os níveis do desenvolvimento do sistema: unitário, integração e sistema. O autor menciona ainda que geralmente o teste de caixa branca é confundido com os testes unitários executados pelos desenvolvedores. Isto é uma visão restrita do teste de caixa branca, já que se trata de algo mais amplo do que o teste de código – é um teste de caminhos. Pode-se utilizá-lo, portanto, não apenas para testar um módulo, como também os caminhos entre os módulos dentro de subsistemas, os subsistemas dentro de um sistema e até mesmo o sistema inteiro (COPELAND, 2004).

O autor descreve quatro desvantagens apresentadas pelas técnicas de teste de caixa branca (COPELAND, 2004):

- o número de caminhos de execução pode ser tão grande que eles não podem ser todos testados;
- os casos de teste escolhidos podem não detectar erros sensíveis à dados;
- os testes de caixa branca assumem que o fluxo de controle está correto ou muito próximo do correto, desta forma, já que os testes são baseados em caminhos existentes, os caminhos não existentes não podem ser descobertos em testes deste tipo;
- o testador deve ter habilidades de programação para entender e avaliar o *software* em teste. No entanto, geralmente a maior parte dos testadores não tem este perfil.

Por outro lado, o autor observa que o teste de caixa branca permite que o testador tenha certeza de que todos os caminhos do *software* em teste foram identificados e testados (COPELAND, 2004).

Dentre as técnicas de teste de caixa preta, o autor apresenta: teste de classe de equivalência (*Equivalence Class Testing*), teste de valores limites (*Boundary Value Testing*), teste de tabela de decisão (*Decision Table Testing*), teste de combinação aos pares (*Pairwise Testing*), teste de transição de estado (*State-Transition Testing*), teste de análise de domínio (*Domain Analysis Testing*) e teste de caso de uso (*Use Case Testing*) (COPELAND, 2004).

O teste de classe de equivalência é uma técnica utilizada para reduzir o número de casos de teste a um tamanho exeqüível mantendo uma cobertura razoável. Segundo Copeland (2004) esta técnica é simples e é utilizada intuitivamente pela maioria dos testadores, embora muitas vezes eles não tenham consciência que se trata de um método formal de desenho do teste.

Uma classe de equivalência consiste de um conjunto de dados que são tratados iguais pelo mesmo módulo ou deveriam produzir o mesmo resultado. Qualquer valor de um dado dentro de uma classe é equivalente, em termos de teste, a outro valor (COPELAND, 2004).

Segundo o autor Copeland (2004), a maior contribuição da classe de equivalência é levar ao teste de valor limite (*boundary value testing*), que é uma técnica também utilizada para reduzir o número de casos de teste a um tamanho exeqüível mantendo uma cobertura razoável. Esta técnica se concentra nos limites pelo fato da maior parte dos defeitos residirem nestes valores. Os testadores experientes, segundo o autor, encontraram esta situação muitas vezes. A técnica consiste em portanto criar casos de teste para cada um dos valores limites escolhendo um ponto no limite, um ponto imediatamente abaixo do limite, e um ponto imediatamente acima do limite (COPELAND, 2004).

De Young e Rätzmann (2003, p. 57-58) descrevem os testes de valores limites como verificações realizadas com os valores de entrada mais extremos em cada contexto. Os campos numéricos são preenchidos com os valores mínimos e máximos, assim como os campos de texto são preenchidos com o número máximo de caracteres possíveis, por exemplo.

Outra técnica de teste de caixa preta descrita por Copeland (2004) é a tabela de decisão (*Decision Table*). As tabelas de decisão são utilizadas para documentar regras de negócio complexas que um sistema deve implementar. Adicionalmente, elas servem como um guia para a criação dos casos de teste. As condições representam várias condições de entrada,

enquanto as ações são os processos que devem ser executados de acordo com as várias combinações das condições de entrada. Cada regra define uma única combinação de condições que resulta na execução de ações associadas àquela regra. Deve-se, segundo Copeland (2004), criar ao menos um caso de testes para cada regra. Se as condições da regra forem binárias, um simples teste para cada combinação é provavelmente suficiente. Por outro lado, se uma condição é uma faixa de valores, deve-se, segundo o autor, considerar testar tanto a extremidade superior quanto a inferior da faixa.

A técnica de teste em pares ou de combinação aos pares (*Pairwise Testing*) se aplica quando o número de combinações de teste é muito grande. Neste caso, ela prega, segundo o Copeland (2004), que ao invés de testar todas as combinações, deve-se testar todos os pares de variáveis, pois isto reduz significativamente o número de testes que devem ser criados e executados. Copeland (2004) descreve que estudos sugerem que a maior parte dos defeitos são ou defeitos de modo simples (*single-mode*) ou de modo duplo (*double-mode*). Os defeitos de modo simples são aqueles onde a função sendo testada simplesmente não funciona e os defeitos de modo duplo são aqueles onde a combinação de uma função ou módulo com outra função ou módulo, formando um par, falha. Esta técnica de teste define um sub-conjunto mínimo que leva ao teste de todos os defeitos "*single*-mode" e "*double-mode*". Copeland (2004) diz que a grande motivação em seu uso é devido ao sucesso em vários projetos, tanto documentados como não documentados.

Os diagramas de transição de estado (*State-transition*) direcionam os esforços de teste através da identificação dos estados, eventos, ações e transições que devem ser testadas. A técnica conhecida como teste de transição de estado (*State-Transition Testing*) recomenda que seja criado um conjunto de casos de teste tais que todas as transições sejam exercitadas ao menos uma vez em teste. Em sistemas de elevado risco, pode-se criar mais casos de teste, abordando todos os caminhos se possível (COPELAND, 2004).

De acordo com Beizer (1990, p. 25) as estratégias de teste de estado são baseadas na utilização de modelos de máquinas de estados-finitos para a estrutura, comportamento ou especificações do comportamento do *software*.

A análise de domínio (*domain analysis*) facilita o teste de múltiplas variáveis simultaneamente. A técnica de teste baseada em análise de domínio (*Domain Analysis*

Testing) é útil pelo fato de raramente se ter tempo para criar casos de teste para cada variável no sistema, já que são muitas. No entanto, freqüentemente as variáveis interagem. Quando o valor de uma das variáveis restringe os valores aceitáveis de uma outra, certos defeitos não podem ser descobertos testando as individualmente. Ele se apóia no teste de classe de equivalência e de valor limite e os generaliza para n dimensões simultâneas. Assim como essas técnicas, procura-se por situações onde o limite foi implementado incorretamente.

O teste de caso de uso (*Use Case Testing*) é uma técnica de teste baseada em casos de uso. Um caso de uso é um cenário que descreve o uso de um sistema por um ator para atingir um objetivo específico. Um ator é um usuário, exercendo um papel em relação ao sistema, procurando utilizar o sistema para realizar algo de valor dentro de um contexto particular. Um cenário é uma seqüência de passos que descrevem as interações entre o ator e o sistema (COPELAND, 2004).

Copeland (2004) ressalta que mesmo criando ao menos um caso de teste para o cenário de sucesso principal e pelo menos um caso de teste para cada extensão, o que provê algum nível de cobertura de teste, está claro que, não importa o quanto seja tentado, a maior parte das combinações de entrada permanecerão não testadas. Não se deve confiar demais na qualidade do sistema neste ponto. Copeland (2004) afirma ainda que o principal componente do teste de transações é o dado de teste. De acordo com Boris Beizer (apud Copeland, 2004), de 30 a 40% dos esforços no teste de transação estão na geração, captura, ou extração dos dados de teste.

Dentre as técnicas de teste de caixa branca o autor descreve em sua obra: teste de fluxo de controle (*Control Flow Testing*) e teste de fluxo de dados (*Data Flow Testing*) (COPELAND, 2004).

O teste de fluxo de controle identifica os caminhos de execução através de um módulo de um programa e então cria e executa os casos de teste para cobrir esses caminhos. Os grafos de fluxo de controle (*Control Flow graphs*) são as fundações do teste defluxo de controle. Os módulos de código são convertidos em grafos, os caminhos através dos grafos são analisados, e os casos de teste são criados a partir da análise. Pelo fato do conjunto de caminhos de base cobrirem todos as bordas e nós (*edges* e *nodes*) do grafo de fluxo de controle, satisfazendo

este critério de teste estruturado, automaticamente se garante a cobertura de base e de ramificação (COPELAND, 2004).

O teste de fluxo de dados (*data flow testing*) se apóia em um erro comum de programação: a referência ao valor de uma variável sem antes atribuir o valor a ela (COPELAND, 2004).

De acordo com Beizer (1990. p. 171-172), o teste de fluxo de dados é o nome dado à família de estratégias de teste baseadas na seleção de caminhos através do fluxo de controle do programa de forma a explorar seqüências de eventos relacionados com o estado dos objetos de dados.

Um grafo de fluxo de dados é similar ao grafo de fluxo de controle de forma que ele mostra o fluxo de processamento através do módulo. Adicionalmente, ele detalha a definição, uso, e destruição de cada uma das variáveis do módulo. Estes diagramas são utilizados para verificar se os padrões de definição, uso e eliminação são apropriados. Copeland (2004) descreve que se deve enumerar os caminhos através de cada módulo e então, para cada variável, criar ao menos um caso de tese para cobrir cada par de "definição e uso".

De Young e Rätzmann (2003, p. 56-58) mencionam uma outra forma diferente de distinguir entre os diversos tipos de teste de acordo com o espectro possível de dados de entrada. Neste contexto, os testes podem ser classificados em: teste de dados randômicos (*Random Data Testing*), teste de verificação de ponto (*Spot Check Testing*) e teste de valores limites (*Boundary Value Testing*).

Os testes de dados randômicos confrontam a aplicação em teste com entradas geradas randomicamente. Os testadores geralmente não prestam atenção ao tipo de dado nestes testes. (DE YOUNG; RÄTZMANN, 2003, p. 56)

Nos testes de verificação de ponto, a componente randômica também está presente, no entanto, os dados são selecionados de uma massa de dados real ao qual o *software* estará sujeito quando estiver em uso. Os dados, no entanto, são selecionados através de um algoritimo randômico e são inseridos no sistema para os testes (DE YOUNG; RÄTZMANN, 2003, p. 56-57).

Já os testes de valores limites foram descritos anteriormente.

Horch (2003, p. 79-82) apresenta outros quatro tipos de testes, os quais, em sua classificação, são considerados testes de tipo especial: regressão (*regression*), estresse (*stress*), recuperação (*recovery*), remoção e restauração (*Back-out and restoration*).

Vale ressaltar que os testes de regressão em algumas obras são considerados uma etapa no ciclo de vida dos testes, tal como já descrito nesta dissertação.

Os testes de regressão visam mostrar que as modificações em uma parte do sistema não invalidaram outras partes. Os testes de regressão são um subconjunto do teste de usuário ou de aceite. Como justificativa Horch (2003, p. 79) cita que vários estudos indicaram que em torno de 50% de todas as mudanças feitas em um *software* resultam na introdução de novos defeitos.

Os testes de estresse cobrem situações em que o *software* é utilizado muito próximo, na própria ou além do limite de sua capacidade requerida (HORCH, 2003, p. 80).

Os testes de recuperação visam validar a recuperação do sistema em condições de término anormais ou paradas no sistema. Estes testes são conduzidos quando uma falha de hardware ou erro de operação danifica o *software* ou os dados (HORCH, 2003, p. 80-81).

Os testes de remoção e restauração são realizados para verificar o sistema em situações em que uma nova versão de um sistema é removida e substituída por uma versão mais antiga, geralmente em situações que a versão apresenta graves problemas em produção, porque provavelmente foi testada insuficientemente. Nestas situações além do sistema ser removido, o banco de dados deve ser restaurado para uma versão compatível com o sistema antigo que será utilizado. O teste requerido neste contexto, segundo Horch (2003, p. 81-82), inclui pelo menos o teste de aceite da versão antiga, que geralmente é acrescido com o mais recente teste de regressão utilizado para a versão antiga do sistema. (HORCH, 2003, p. 81-82)

De Young e Rätzmann (2003, p. 43-49) citam outros tipos de testes, como os exploratórios, pela utilização (*testing by using*), de fumaça (*smoke tests*), embutido (*embedded testing*), teste ao vivo (*live testing*), de segurança (*security testing*).

O objetivo do teste exploratório é encontrar pontos de falhas críticas no *software* o mais rápido possível. De Young e Rätzmann (2003, p. 45) defendem que demandaria muito tempo, e acima de tudo, seria uma perda de tempo, conhecer inicialmente a aplicação, planejar os casos de teste, e finalmente executar cada caso de teste na seqüência. Ele afirma que no espaço de tempo em que o testador conhecer o suficiente a aplicação para encontrar e escrever todos os casos de teste, a próxima versão do programa já está sendo finalizada. (DE YOUNG; RÄTZMANN, 2003, p. 45)

De Young e Rätzmann (2003, p. 45) apresentam ainda um "subconjunto" do teste exploratório – o teste de guerrilha (*guerilla testing*). Esta técnica envolve um teste curto, porém intensivo, de uma seção limitada de um programa por um testador experiente. O objetivo é avaliar o grau de risco trazido por esta unidade do programa. Utilizando os meios mais extremos disponíveis, o testador tenta fazer o programa falhar ou abortar. Se o programa resistir ao "ataque" razoavelmente bem, a unidade "atacada" pode ser colocada de volta para os testes adicionais ou mesmo pode ser deixada de fora do plano de teste. (DE YOUNG; RÄTZMANN, 2003, p. 45)

O teste pela utilização (*testing by using*) é considerado por De Young e Rätzmann (2003, p. 43) uma das estratégias de teste mais frutíferas, pois ela busca responder questionamentos surgidos durante um processo de desenvolvimento adaptativo que somente podem ser respondidos através do uso, como (DE YOUNG; RÄTZMANN, 2003, p. 43):

- O *software* é realmente útil? Como a usabilidade pode ser melhorada? Quais funcionalidades estão faltando para deixá-lo realmente excelente?
- O *software* suporta todos os fluxos de trabalho ou pelo menos os mais importantes? O *software* melhora ou impede o fluxo de trabalho?
- Os usuários enxergam o seu domínio de trabalho refletido no *software*? Os termos especializados estão corretos e as abstrações lógicas?
- É interessante trabalhar com o *software* ou é enfadonho?

Os testes de fumaça (*smoke tests*) são uma bateria de testes que verificam as funcionalidades básicas do programa. O nome se originou da fumaça, no sentido metafórico, que sai do computador quando o *software* falha em um destes testes. Se o *software* falhar no teste de

fumaça, ele não é aceitável e não é liberado para os demais testes. Freqüentemente estes testes são automatizados. Eles pertencem a categoria dos testes positivos – eles buscam provar que uma propriedade funcional específica existe e que nenhum erro surgiu (no caso de uma nova versão). Os testes de fumaça são considerados como um subconjunto dos testes de regressão (DE YOUNG; RÄTZMANN, 2003, p. 46-47).

Os testes embutidos são testes que podem ser embutidos no código fonte do *software* para a realização de validações. Algumas linguagens possuem instruções ASSERT (ou similares) para testar as precondições de uma rotina ou métodos de uma classe. Este conceito é conhecido como "*Design by Contract*" na linguagem Eiffel. Neste caso, os testes são diretamente integrados à linguagem (DE YOUNG; RÄTZMANN, 2003, p. 47-48).

O teste ao vivo de um sistema é conduzido por usuários em perspectiva, usuários iniciais, especialistas, desenvolvedores interessados no *software*, e assim por diante. A forma mais conhecida do teste ao vivo é o beta-teste feito antes da liberação oficial de um produto. Segundo o autor, os resultados de um teste ao vivo são freqüentemente muito vagos. No entanto, o teste ao vivo freqüentemente rende informações surpreendentemente importantes como, por exemplo, que o programa está sendo utilizado de formas não planejadas pelos desenvolvedores (DE YOUNG; RÄTZMANN, 2003, p. 48-49).

Os testes de segurança determinam se o programa e seus dados são protegidos de acessos hostis, sejam ataques externos ou internos, nos quais usuários tentam acessar informações sem a devida autorização (DE YOUNG; RÄTZMANN, 2003, p. 55).

2.6.13 Teste apoiado em taxonomias

O *Software Engineering Institute* publicou um documento intitulado "*Taxonomy-Based Risk Identification*" que pode ser utilizado para identificar, classificar, e avaliar os diferentes fatores de risco encontrados no desenvolvimento de *software*, conforme é descrito por Copeland (2004). A Tabela 28 apresenta uma tabela contendo esta taxonomia.

Tabela 28- A taxonomia de identificação de risco do SEI

Classe	Elemento	Atributo
Engenharia do produto	Requisitos	Estabilidade

		Completude
		Claridade
		Validade
		Factibilidade
		Precedente
		Escala
		Funcionalidade
		Dificuldade
	Modelagem	Interfaces
		Performance
		Testabilidade
		Factibilidade
	Codificação e teste unitário	Teste
		Codificação/Implementação
		Ambiente
	Integração e teste	Produto
		Sistema
		Manutenibilidade
		Confiabilidade
	Especialidades de engenharia	Segurança (*safety*)
		Segurança (*security*)
		Fatores humanos
		Especificações
		Formalidade
	Processo de desenvolvimento	Conveniência
		Controle de processo
		Familiaridade
		Controle de produto
		Capacidade
		Conveniência
		Usabilidade
	Sistema de desenvolvimento	Familiaridade
		Confiabilidade
		Suporte ao sistema
Ambiente de desenvolvimento		"Entregabilidade"
		Planejamento
	Processo gerencial	Organização do projeto
		Experiência em gerenciamento
		Interfaces do programa
		Monitoramento
	Métodos gerenciais	Gerenciamento pessoal
		Garantia da qualidade
		Gerencia de configuração
		Atitude de qualidade
	Ambiente de trabalho	Cooperação
		Comunicação
		Moral
Restrições do programa	Recursos	Cronograma
		Staff
		Orçamento

Contrato	Facilidades
	Tipos de contrato
	Restrições
	Dependências
Interfaces do programa	Clientes
	Contratadores associados
	Subcontratadores
	Contratadores principais
	Gerenciamento corporativo
	Vendors
	Políticas

FONTE: COPELAND, 2004

Copeland (2004) descreve que um testador deve estressar certos tipos de testes de acordo com os elementos e atributos que podem representar os maiores riscos. Um exemplo desta relação é mostrado na Tabela 29.

Tabela 29 - Relação entre foco de preocupação e atividade que deve ser enfatizada

Foco da preocupação	**Ênfase**
Estabilidade dos requisitos	Rastreabilidade formal
Requisitos incompletos	Testes exploratórios
Requisitos escritos de forma imprecisa	Tabelas de decisão ou diagramas de transição de estados
Dificuldade na realização da modelagem	Teste de controle de fluxo
Performance do sistema	Teste de performance
Falta de teste unitário	Recursos de teste adicionais
Problemas de usabilidade	Testes de usabilidade

FONTE: COPELAND, 2004

Copeland (2004) também apresenta a taxonomia das características da qualidade definidas pela norma ISO 9126 (*Software Product Evaluation—Quality Characteristics and Guidelines*), que foca na medida da qualidade dos sistemas de *software*. Esta norma, tal como é descrito por Copeland (2004), define a qualidade do *software* em termos de seis características principais e 21 sub-características e define um processo para avaliar cada uma delas. A Tabela 30 apresenta esta taxonomia de atributos da qualidade.

Tabela 30 – A taxonomia das características de qualidade da ISO 9126

Características da qualidade	**Sub-características**
Funcionalidade (As funções requeridas estão diponíveis no *software*?)	Conveniência
	Acurácia
	Interoperabilidade
	Segurança (*security*)
Confiabilidade (Quão confiável é o *software*?)	Maturidade
	Tolerância a falhas

Usabilidade (O *software* é fácil de utilizar?)	Recuperabilidade Compreensibilidade Aprendibilidade Operabilidade Atratividade
Eficiência (Quão eficiente é o *software*?)	Comportamento temporal Comportamento dos recursos
Manutenabilidade (Quão fácil é modificar o *software*?)	Analisibilidade Modificabilidade Estabilidade Testabilidade
Portabilidade (Quão fácil é transferor o *software* para outro ambiente operacional?)	Adaptabilidade Instabilidade Coexistência Substituibilidade

FONTE: COPELAND, 2004

Cada uma destas características e sub-características sugerem áreas de risco e, portanto, áreas para as quais testes devem ser criados. Uma avaliação da importância destas características deve ser realizada primeiramente de forma que o nível apropriado de teste possa ser executado. Copeland (2004) recomenda que um processo similar "se você está preocupado com / você deve enfatizar" deve ser utilizado baseado na taxonomia da norma ISO 9126. Ele afirma ainda que esta taxonomia no nível de projeto pode ser utilizada para guiar os testes em um nível estratégico, enquanto que para ajudar no desenho dos testes pode-se utilizar as taxonomias de defeito de *software*.

2.7 Métricas

Segundo Horch (2003), as métricas são uma parte essencial de qualquer sistema de qualidade de *software*, já que elas permitem que as medidas se tornem informações para o gerenciamento da qualidade. Para o autor, as métricas somente podem ser consideradas úteis quando servem de base informacional para os decisores em um processo de desenvolvimento de *software*.

Horch (2003) adota a definição de que uma métrica é uma relação entre duas medidas.

A análise de defeito é a disciplina de mensurar o desenvolvimento e o uso do *software*. As medidas podem vir de questionários, dados de defeitos, cronogramas e orçamentos, características do sistema, utilização de suporte, e assim por diante. Cada organização deve, segundo Horch (2003), descobrir e determinar as fontes de medidas que melhor se encaixam às suas necessidades e contexto.

As métricas de defeito são aquelas compostas por medidas relacionadas com os defeitos. As métricas modem incluir número de falhas no *software*, número de chamadas feitas ao suporte, tempo gasto no registro após uma falha, e custo das vendas perdidas (HORCH, 2003).

Os objetivos do sistema de qualidade de *software* lidam com questões relacionadas com defeitos e seus efeitos. Estas questões, por sua vez, lidam com métricas que podem prover informações de defeitos. As métricas então lidam com medidas que devem ser coletadas (HORCH, 2003).

Horch (2003) afirma que é possível criar um número infinito de métricas combinando-se duas medidas quaisquer e que a seleção de métricas específicas e relevantes é um dos objetivos do sistema de qualidade de *software* (HORCH, 2003).

Uma vez escolhidas as métricas, é importante prestar atenção na coleta das medidas necessárias para as métricas. Do ponto de vista da garantia da qualidade, a maior parte das métricas está relacionada com os defeitos (HORCH, 2003).

Horch (2003, p. 107-111) ressalta que as métricas podem ser orientadas para o produto ou para o processo.

Para um dado *software*, os defeitos podem servir como uma indicação de seu progresso em ficar pronto para a implantação. Algumas métricas podem ajudar na identificação de módulos ou subsistemas propenso a erros. Outras indicam a redução na detecção de defeitos a medida que eles são encontrados e removidos. (HORCH, 2003, p. 107-111)

O retrabalho é geralmente considerado como qualquer trabalho que é refeito pelo fato de não ter sido feito corretamente da primeira vez. As causas mais freqüentes de retrabalho são as correções necessárias para resolver os defeitos ou incompatibilidades com padrões. O

monitoramento de métricas de retrabalho pode ajudar os profissionais de garantia da qualidade de *software* a demonstrar a conveniência de um melhor planejamento de projeto e uma atenção maior aos requisitos (HORCH, 2003).

A experiência com defeito de longo prazo ajuda a entender o processo de desenvolvimento e sua estabilidade, previsibilidade, e nível de melhoria. O profissional de garantia da qualidade pode acompanhar as tendências na detecção e na correção de defeitos como indicadores da maturidade do processo. Métricas de produtividade dão indicadores de efetividade do processo, qualidade das técnicas de estimativa, qualidade das técnicas de detecção de defeitos etc. Algumas são baseadas em dados de defeitos enquanto outras não (HORCH, 2003, p. 107-111).

Análise de tendência é a comparação de medidas e métricas para determinar como um processo está se comportando. O controle estatístico de processo, taxas de detecção de erros, saídas em relação ao tempo, custo da qualidade, e utilização do suporte são exemplos de métricas que podem ser utilizadas ao longo de um período de tempo. Tais estudos olham para as tendências e descrevem o processo de desenvolvimento ou suas reações à mudanças intencionais. O uso de cartas de controle de processo (process control charts) para *software* pode ajudar a descrever o comportamento do processo de desenvolvimento, tal como já descrito. As cartas podem também ajudar a identificar as mudanças no comportamento do processo em resposta à mudanças intencionais feitas no processo (HORCH, 2003, p. 107-111).

Horch (2003) apresenta a métrica de custo da qualidade (*cost of quality* - COQ) utilizada para mensurar o valor do sistema de qualidade de *software*. Este custo calculado pela soma do custo para atingir a qualidade (*cost of achieving quality* – COA) com o custo da falha (*cost of failure* - COF). A Tabela 31 apresenta os componentes do custo da qualidade (HORCH, 2003, p. 107-111).

Tabela 31 – Custos da Qualidade

CATEGORIAS DOS CUSTOS DA QUALIDADE	ITEM QUE CONTRIBUI COM OS CUSTOS DA CATEGORIA
Prevenção	Treinamento e educação CM Planejamento

CATEGORIAS DOS CUSTOS DA QUALIDADE	ITEM QUE CONTRIBUI COM OS CUSTOS DA CATEGORIA
	Normas
Avaliação	Revisões (até encontrar defeito) Teste (até encontrar defeito)
Falha	Re-trabalho Re-escrever especificações e planos Revisar novamente (após a correção do defeito) Re-testar (após a correção do defeito) Sucateamento Gerenciamento do problema com o cliente Lost customers Oportunidades perdidas

FONTE: HORCH, 2003, p. 108

Horch (2003) também sugere algumas métricas orientadas à metas do sistema de qualidade de software mostradas na Tabela 32.

Tabela 32 - Objetivos e Métricas

Objetivo do SQS	Métrica aplicável [20]
Melhoria no gerenciamento de defeitos	COQ de mudanças/cronograma de implementação de SQS Custo do *software* rejeitado (sucata)/custo total do projeto Custo das correções de defeito/custo da detecção de defeito Densidade de defeito/produto Densidade de defeito/fase no ciclo de vida Defeitos encontrados pelas revisões/defeitos encontrados pelo teste Defeitos detectados pelo usuário/defeitos detectados pelo desenvolvedor CR fechadas/total de CR abertas CR abertas remanescentes/ CR fechadas CR abertas e fechadas/período de tempo Tempo médio entre falhas Confiabilidade do produto Chamadas de suporte/produto
Melhoria nos requisitos	Requisitos mudados/total de requisitos Requisitos implementados/total de requisitos Erros de requisitos/total de erros
Melhoria na detecção de defeitos	Testes executados com sucesso/total de testes planejados Defeitos encontrados nas revisões/defeitos encontrados no teste Densidade de defeito/produto Defeitos insetidos por fase no ciclo de vida/defeitos detectados por fase no ciclo de vida Defeitos detectados pelo usuário/defeitos detectados pelo desenvolvedor
Melhoria na produtividade do desenvolvedor	KLOC ou pontos de função/pessoa em um mês Cronograma ou orçamento real/estimado Gastos do orçamento/status do cronograma Tempo médio para reparar um defeito Correções incorretas/total de correções Defeitos do produto/complexidade do produto
Melhoria nas técnicas de	Cronograma ou orçamento real/estimado

[20] Algumas métricas ou termos não foram traduzidos por gerar um prejuízo no entendimento ou por serem conhecidas e utilizadas de tais formas no mercado.

Objetivo do SQS	Métrica aplicável [20]
estimativa	Tempo médio para reparar um defeito Gastos do orçamento/status do cronograma
Melhoria crescente	Correções incorretas/total de correções Tempo médio de reparar um defeito Defeitos detectados pelo usuário/defeitos detectados pelo desenvolvedor

FONTE:HORCH, 2003, p. 109

Os defeitos têm um papel importante nas métricas de qualidade de *software*, já que a maior parte destas está relacionadas àqueles. Desta forma, Horch (2003) recomenda a classificação dos defeitos, pois segundo o autor, permite guiar a solução deles bem como identificar fraquezas no processo de desenvolvimento de *software* e prever possíveis áreas de problemáticas no futuro. De acordo com Horch (2003) os defeitos podem ser classificados de acordo com suas várias características, dentre elas (HORCH, 2003, p. 110-111):

- Severidade do defeito se encontrado em operação
- Prioridade de reparo imediato
- Origem (fase no ciclo de vida) do defeito
- Tipo de defeito
- Fase (fase no ciclo de vida) na qual o defeito foi encontrado
- Método pelo qual o defeito foi encontrado
- Custo estimado e real para reparar o defeito

Outras métricas, não relacionadas à classificação dos defeitos, apresentadas por Horch (2003, p. 110-111) são: número de defeitos, freqüência dos defeitos, solicitações de mudanças e correções abertas e resolvidas, tempo entre a detecção dos defeitos, defeitos resultantes de correções prévias de defeitos; tamanho da mudança, relatórios de defeito incorretos (informações incorretas nas solicitações de mudanças e correções)

Horch (2003, p. 110-111) ressalta que nem todos os defeitos são defeitos, pois em alguns casos uma operação detectada no sistema não está incorreta e sim, apenas, não era esperada, ou seja, o caso de teste pode conter um resultado esperado incorreto ou o usuário não possui experiência. Nestes casos, a produtividade pode ser prejudicada na tentativa de corrigir defeitos baseado em relatórios incorretos. Horch (2003, p. 110-111) recomenda treinar os usuários ou os responsáveis por reportar os defeitos, ou mesmo talvez melhorar a documentação do usuário.

Os dados de defeito apenas não são suficientes para a garantia da qualidade. Outras métricas não relacionadas a defeitos também são utilizadas como tamanho do projeto, orçamento e cronograma, tempo de processamento, número de pessoas envolvidas na atividade, e etc. Estas métricas podem ser obtidas diretamente e nenhuma interpretação delas é geralmente necessária (HORCH, 2003, p. 110-111).

No caso ainda da qualidade de *software*, algumas medidas não estão disponíveis no formato numérico e sim dependem da quantificação de dados subjetivos, tais como: impressões do cliente, qualidade percebida em alguma escala subjetiva, estimativas de qualidade, e etc. Estas medidas devem ser utilizadas com cuidado já que geralmente não há formas precisas de quantifica-las ou validá-las (HORCH, 2003, p. 110-111).

2.8 Normas e Padrões

Diversas são as origens das normas. Elas podem ser internas a uma organização, ou seja, desenvolvidas internamente em uma empresa, ou externas. As fontes externas são (HORCH, 2003, p. 45):

- normas internacionais - ISO/IEC;
- grupos de profissionais e da indústria - IEEE, *Electronic Industries Association* (EIA), *American National Standards Institute* (ANSI), *American Society for Quality* etc;
- agências governamentais - *Department of Defense* (DoD), *Nuclear Regulatory Agency*, *National Institute of Standards and Technology* (NIST) etc;
- grupos de usuários de empresas - GUIDE *International*, SHARE (IBM), DECUS (DEC) etc.

Horch (2003, p.45) menciona que uma das vantagens de utilizar normas externas é que elas refletem o consenso do grupo envolvido. A Tabela 33 traz uma série de normas de diversas origens disponíveis ligadas ao desenvolvimento de *software*.

Tabela 33 – Fontes representatives de normas

Assunto principal	Área específica	Criador	Número da norma
Ciclo de vida de *Software*	Processos de ciclo de vida	IEEE ISO/IEC	1074, 1074.1 12207
	Gerenciamento de projeto	IEEE	1058
	Desenvolvimento	DoD IEEE ISO	498 1074 12207
	Revisões	IEEE NIST	1028, 1059 500-165
	Teste	IEEE NIST	829, 1008, 1012,1059 500-75, 500-165
	Programa de qualidade	IEEE AS ISO NRC	1298 3563.1, 3563.2 9000 *et al* NUREG/CR-4640
	Métricas	IEEE ISO/IEC	982.1, 982.2, 1044, 1044.1, 1045, 1061 9126
	Ferramentas *Case*	IEEE	1175, 1209, 1343
Documentação	Planos de qualidade	IEEE IEEE/EIA	730, 730.1 1498/IS 640
	Especificações de requisitos	IEEE ISO/IEC	830 12207
	Especificações de modelagem	IEEE ISO/IEC	1016, 1016.11 12207
	Documentação do usuário	IEEE	1063
Nomenclatura	CM – gerenciamento de mudanças	IEEE EIA	1042, 828 649
User development	Pacotes de *Software*	ISO/IEC	12119

FONTE: HORCH, 2003, p. 45

A Tabela 34 apresenta uma lista de uma série de normas do IEEE para Engenharia de *Software*.

Tabela 34 - Normas IEEE para Engenharia de *Software*

Norma	Descrição
610.12-1990(R2002)	*IEEE Standard Glossary of Software Engineering Terminology*
730-2002	*IEEE Standard for Software Quality Assurance Plans*
828-1998	*IEEE Standard for Software Configuration Management Plans*
829-1998	*IEEE Standard for Software Test Documentation*
830-1998	*IEEE Recommended Practice for Software Requirements Specifications*
982.1-1988	*IEEE Standard Dictionary of Measures to Produce Reliable Software*
1008-1987 (R1993, R2002)	*IEEE Standard for Software Unit Testing*
1012-1998	*IEEE Standard for Software Verification and Validation*
1012a-1998	*IEEE Standard for Software Verification and Validation - Supplement to 1012-1998 - Content Map to IEEE 12207.1*

1016-1998	*IEEE Recommended Practice for Software Design Descriptions*
1028-1997(R2002)	*IEEE Standard for Software Reviews*
1044-1993(R2002)	*IEEE Standard Classification for Anomalies*
1045-1992, (R2002)	*IEEE Standard for Software Productivity Metrics*
1058-1998	*IEEE Standard for Software Project Management Plans*
1058.1-1987 (R1993)	*IEEE Standard for Software Project Management Plans*
1061-1998	*IEEE Standard for Software Quality Metrics Methodology*
1062, 1998 *Edition* (R2002)	*IEEE Recommended Practice for Software Acquisition (includes IEEE 1062a)*
1063-2001	*IEEE Standard for Software User Documentation*
1074-1997	*IEEE Standard for Developing Software Life Cycle Processes*
1175.1-2002	*IEEE Guide for CASE Tool Interconnections-Classification and Description*
1219-1998	*IEEE Standard for Software Maintenance*
1220-1998	*IEEE Standard for the Application and Management of the Systems Engineering Process*
1228-1994 (2002)	*IEEE Standard for Software Safety Plans*
1233, 1998 *Edition* (R2002)	*IEEE Guide for Developing System Requirements Specifications (including IEEE 1233a)*
1320.1-1998	*IEEE Standard for Functional Modeling Language - Syntax and Semantics for IDEF0*
1320.2-1998	*IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X97 (IDEFobject)*
1362-1998	*IEEE Guide for Information Technology-System Definition -Concept of Operation Document*
1420.1-1995 (R2002)	*IEEE Standard for Information Technology--Software Reuse--Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM)*
1420.1a-1996 (R2002)	*IEEE Supplement to Standard for Information Technology--Software Reuse--Data Model for Reuse Library Interoperability: Asset Certification Framework*
1420.1b-1999 (R2002)	*IEEE Supplement to IEEE Standard for Information Technology--Software Reuse--Data Model for Reuse Library Interoperability: Intellectual Property Rights Framework*
1465-1998 (12119:1998 ISO/IEC)	*Information Technology - Software Packages - Quality Requirements and Testing*
1471-2000	*IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*
1490-1998	*IEEE Guide (©IEEE) - Adoption of PMI Standard- A Guide to the Project Management Body of Knowledge(©PMI)*
1517-1999	*IEEE Standard for Information Technology - Software Life Cycle Processes - Reuse Processes*
2001-1999	*IEEE Recommended Practice for Internet Practices--Web page Engineering--Intranet/Extranet Applications*
1540-2001	*IEEE Standard for Software Life Cycle Processes-Risk Management*
J-Std-016-1995	*EIA/IEEE Interim Standard for Information Technology--Software Life Cycle Processes--Software Development Acquirer-Supplier Agreement (Issued for Trial Use)*
12207.0-1996	*IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology--Software Life Cycle Processes.*
12207.1-1997	*IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology-- Software Life Cycle Processes--Life cycle data*
12207.2-1997	*IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology-- Software Life Cycle Processes--Implementation considerations*

14143.1-2000 *Implementation Note for IEEE Adoption of ISO/IEC 14143-1:1998, Information Technology-Software Measurement-Functional Size Measurement-Part 1: Definition of Concepts*

FONTE: Standards IEEE para Ingeniería del *Software* (DOCUMENTO SEPARADO)

2.8.1 Tempo investido nos testes de *software*

Fewster e Graham (1999) afirmam que na época em que o livro foi publicado, final da década de 70, uma regra bem conhecida era aquela em que 50% do tempo e um pouco mais de 50% do custo total eram gastos no teste do *software* sendo desenvolvido. Vinte e cinco anos depois, segundo os autores, mesmo com novos métodos de desenvolvimento, ferramentas e linguagens, o teste ainda tem um papel muito importante em qualquer projeto de *software*.

O esforço geralmente alocado para o teste e revisão (documentação, modelos e conceitos) é em torno de 30 a 90% do esforço total de projeto. Estes números estão em crescimento, colocando o teste e a revisão como consumidores de 50% do esforço total do projeto de acordo com o NIST (DE YOUNG; RÄTZMANN, 2003, p. 224).

Os testes unitários chegam a consumir de 8 a 35% do esforço total de projeto, variando de acordo com a complexidade, ou seja, quanto maior a complexidade, maior é este número (DE YOUNG; RÄTZMANN, 2003, p. 224)

Os testes de integração aumentam com a complexidade do sistema, variando de 5% do esforço total de projeto para sistemas simples até 30% em sistemas complexos (DE YOUNG; RÄTZMANN, 2003, p. 225).

Os testes de sistema seguem a mesma lógica de crescimento de acordo com a complexidade do sistema, variando de 5 a 25% do esforço total de projeto (DE YOUNG; RÄTZMANN, 2003, p. 225).

O esforço alocado aos testes de regressão estão relacionados com os métodos de teste empregados. Se o teste de regressão é totalmente manual, pode ser facilmente dispendido 100% do tempo do teste inicial para cada *release* principal. Segundo a experiência dos autores, os testes automatizados de regressão requerem em torno de 30% do esforço dos testes

manuais para os principais *releases*, podendo este valor ser reduzido substancialmente empregando-se uma modelagem dos componentes do *software* orientada a testes (DE YOUNG; RÄTZMANN, 2003, p. 225).

O esforço requerido para o teste de usuário ou de aceite pode variar entre 10 e 30% do esforço total de projeto, aumentando de acordo com a complexidade deste.

De Young e Rätzmann (2003, p. 225-226) sugerem a seguinte distribuição de esforços, em relação ao total do projeto, nos desenvolvimentos de *software* de complexidade baixa para moderada, com diversos módulos e com elevada complexidade tecnológica:

- 5% para os testes unitários;
- 15% para os testes de integração;
- 10% para os testes de sistema;
- 20% para os testes de usuário.

Os autores De Young e Rätzmann (2003, p. 225-226) ressaltam que se o *software* a ser desenvolvido não é para um uso sério, estes esforços podem ser reduzidos.

2.8.2 Eficiência na remoção de defeitos dos testes de *software*

Baseado em um estudo especial encomendado pelo departamento de defesa dos EUA, Jones (IN)) (*Software Productivity Research*, 1994; Jones, 2000) estimou a taxa de eficiência na remoção de defeitos nas organizações de acordo com os diferentes níveis do CMM (KAN, 200, p. 181):

- Nível 1: 85%
- Nível 2: 89%
- Nível 3: 91%
- Nível 4: 93%
- Nível 5: 95%

Os percentuais acumulados de defeitos removidos pelo teste de aceite (a ultima fase antes da liberação do produto) por fase de inserção, para o CMMI nivel 4, é mostrada na Tabela 35. Baseando-se em dados históricos e recentes de três organizações de engenharia de *software*, Diaz e King (2002 apud Kan, 2003, p. 181) relataram que eficácia de contenção desta fase pro nível de CMM é (KAN, 2003, p. 181):

- Nível 2: 25.5%
- Nível 3: 41.5%
- Nível 4: 62.3%
- Nível 5: 87.3%

Tabela 35 – Percentuais acumulados de defeitos removidos por fase para o CMMI nível 4

Fase de inserção	% acumulado dos defeitos removidos pelo teste de aceite
Requisitos	94%
Modelagem alto nível	95%
Modelagem detalhada	96%
Teste unitário e codificação	94%
Teste de integração	75%
Teste de sistema	70%
Teste de aceite	70%

FONTE: KAN, 2003, p. 181

De acordo com Jones (2000), citado por Kan (2003, p. 182), em geral, muitas formas de teste possuem uma eficiência menor que 30%. A eficiência acumulada de uma seqüência de estágios de teste, no entanto, pode atingir 80% (KAN, 2003, p. 182).

Um relatório de 1995 de Capers Jones (JONES, 1994 apud HUTCHESON, 2003, p. 25) denominado "*Software Quality for 1995: What Works and What Doesn't*" relatou a performance de 4 técnicas de remoção de defeitos utilizadas individualmente (HUTCHESON, 2003, p. 25):

- Inspeções formais de modelagem: 45%-68%;
- Testes formais de *software*: 37%-60%;
- Garantia da qualidade formal: 32%-55%;
- Nenhum método formal: 30%-50%.

Quando combinadas, as inspeções formais de modelagem e de código possuem uma eficiência entre 70 e 90%. Entretanto, a melhor combinação encontrada no estudo é composta pelas inspeções formais de modelagem, garantia da qualidade formal e teste formal, resultando em uma eficiência entre 77 e 95% (HUTCHESON, 2003, p. 25).

A Tabela 36 mostra o custo relativo de remoção de defeitos ao longo das fases de desenvolvimento de software, de acordo com Dustin (2003).

Tabela 36 - Custo relativo da remoção de defeitos

Fase no ciclo de vida de desenvolvimento	Custo relativo de correção (US$)
Definição	1
Modelagem alto nível	2
Modelagem baixo nível	5
Codificação	10
Teste unitário	15
Teste de integração	22
Teste de sistema	50
Pós entrega	100+

FONTE: DUSTIN, 2003

Kan (2003) menciona que em sistemas grandes, as revisões podem reduzir o número de erros que chegam às fases de teste em um fator de 10, e estas reduções geram uma economia de 50 a 80% nos custos do teste. O autor menciona ainda que o custo da remoção de defeitos nas fases de inspeção de modelo e código, teste e operação pelo usuário, baseado em dados do laboratório da IBM em Santa Teresa (Califórnia), seguem a proporção de 1/20/82. A razão encontrada com dados da IBM de Rochester mostraram a proporção de 1/13/92.

3 METODOLOGIA DA PESQUISA

3.1 Introdução

Este capítulo apresenta a metodologia empregada na modelagem da pesquisa e no desenvolvimento das atividades de campo. Refere-se também à metodologia utilizada no tratamento de dados, na formulação das conclusões e na redação do texto final. Inicialmente, a estratégia empregada na pesquisa e o seu propósito são apresentados. A seguir, o *design* de pesquisa é explicitado: a composição das partes do instrumento de pesquisa, são apontadas as fontes de dados e de informações empregadas e a validade e fidedignidade da pesquisa são comentadas. No tópico seguinte os procedimentos de campo são detalhados para o acesso às empresas, o acesso aos dados e às informações e também para a coleta de dados e de informações. Depois, a próxima seção descreve o tratamento de dados e de informações e na seguinte a formulação das conclusões é delineada. Finalmente, detalhes sobre a composição do relato do estudo de caso são evidenciados e a figura-síntese da pesquisa é apresentada.

3.2 Estratégia e propósito da pesquisa

A fim de estudar a utilização das práticas de teste de *software*, os atributos de qualidade conhecidos e considerados pelas empresas, bem como avaliar a qualidade dos processos de teste utilizados foi empregada uma pesquisa qualitativa aplicada através de um questionário estruturado em três partes, onde cada uma procura endereçar a coleta de dados para cada uma das informações descritas. Essa estratégia se revelou adequada para tratar o problema em questão uma vez que envolve o levantamento de uma série de informações ligadas que podem ser mapeadas em perguntas fechadas, tornando mais fácil uma análise estatística e a consolidação das respostas para a descrição dos fenômenos. Quanto ao propósito de pesquisa, é necessário que seja descritiva.

3.3 Modelagem da pesquisa

3.3.1 O instrumento de pesquisa

Dada a proposta de pesquisa tornou-se necessário construir um instrumento de pesquisa composto quase que em sua totalidade por perguntas fechadas de múltipla escolha e estruturado em três partes (Figura 39).

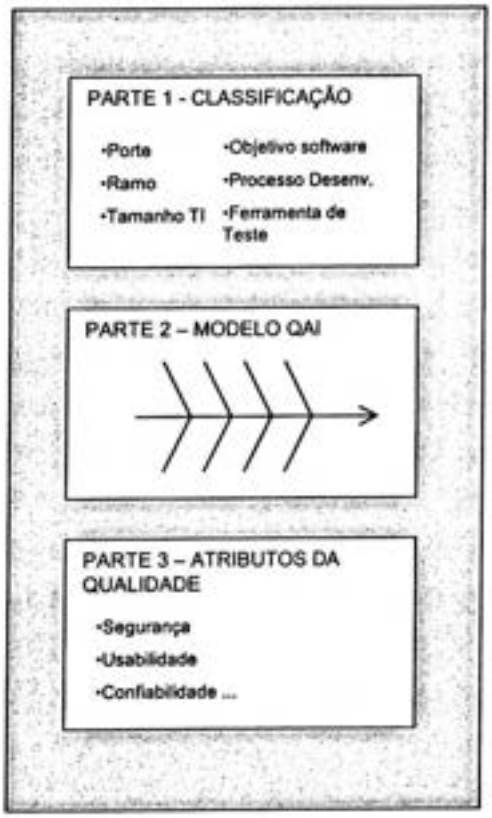

Figura 39 - Representação esquemática do instrumento de pesquisa

A primeira parte visa coletar informações para classificar a empresa respondente, tais como o seu porte, ramo de atuação, realização de desenvolvimento de *software*, contratação de outras empresas para o desenvolvimento de *software*, tipo de processo de desenvolvimento de *software*, existência de processos de testes e validações, utilização de ferramentas de testes, tipos de ferramentas utilizadas, intenção em utilizar ferramentas, tamanho da área de TI, objetivo dos *softwares* desenvolvidos, percepção de qualidade dos respondentes em relação aos *softwares* desenvolvidos, bem como a percepção de qualidade dos usuários na opinião dos respondentes e etc.

A segunda parte visa aplicar o modelo desenvolvido pelo QAI (*Quality Assurance Institute*) e apresentado por William E. Perry em sua obra "*Effective Methods for Software Testing*" para a avaliação da qualidade dos processos de testes existentes nas empresas. Este modelo é baseado em estudos realizados pelo QAI para entender o que torna as organizações que testam *software* bem sucedidas. Como resultado o QAI identificou oito critérios que são normalmente associados às organizações de teste de classe mundial. Estes oito critérios são o planejamento do teste, treinamento dos testadores, suporte gerencial aos testes, satisfação dos usuários com os testes, utilização de processos de testes, práticas eficientes de testes, utilização de ferramentas de testes, e controle de qualidade sobre os processos de teste.

Segundo o QAI, quando estes oito critérios estão presentes e funcionando o resultado é uma organização de teste de classe mundial. O modelo de avaliação do QAI possui cinco áreas para endereçar cara um dos oito critérios, que para efeitos desta pesquisa, algumas destas áreas foram desdobradas em mais de uma pergunta no questionário. Quanto mais estiverem presentes estas áreas, bem como estiverem funcionando, mais esta categoria contribuirá, de acordo com o modelo, para um teste de classe mundial. A Figura 40 mostra um diagrama de causa-efeito indicando as áreas, chamadas de *drivers*, que resultam, dentro do modelo apresentado pelo QAI em uma organização de teste de classe mundial (PERRY, 2000).

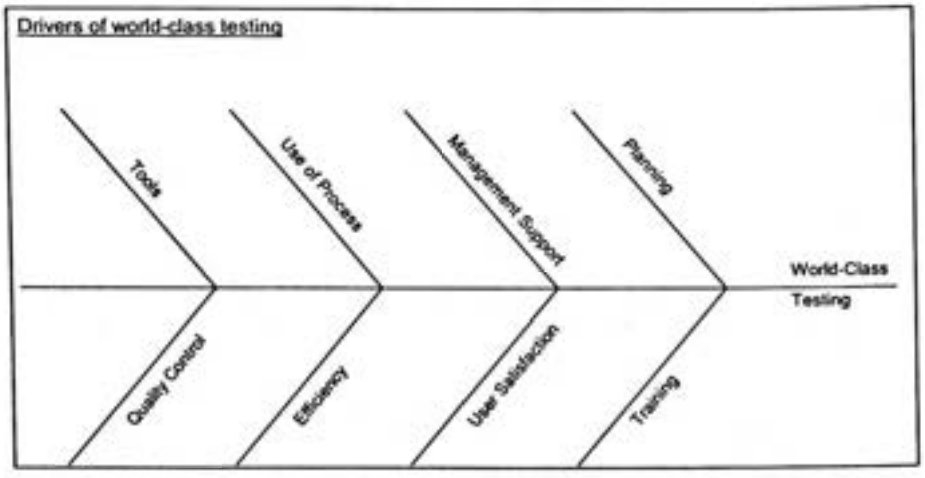

Figura 40 - Visão geral da avaliação do processo de teste (QAI)

Segundo Perry (2000), o resultado desta avaliação pode ser utilizado de uma das seguintes maneiras:

- Determinar o status atual do teste de *software* de uma organização versus uma organização de teste de classe mundial. As respostas irão indicar as áreas fortes e fracas comparadas com uma organização de teste de classe mundial;
- Desenvolver um objetivo para o teste de *software* se tornar um teste de classe mundial. O modelo de classe mundial do QAI, segundo PERRY, indica um perfil de uma organização de teste de classe mundial. Atingir este perfil pode ser o objetivo para a área de teste;
- Desenvolver um plano de melhoria.

Dentro do escopo deste trabalho, o objetivo da aplicação deste modelo de avaliação na pesquisa é o primeiro, ou seja, entender o quanto as organizações pesquisadas, dentro de uma média, encontram-se compatíveis com uma organização de teste de classe mundial, bem como determinar as áreas fortes e fracas.

A terceira parte do instrumento de pesquisa se destina a avaliar o grau de conhecimento de uma série de atributos de qualidade de *software* presentes em diversos modelos, como ISO/IEC 9126 e Mc Call, por exemplo, bem como a realização de testes e validações por parte das empresas para avaliar cada um destes. Dentro do instrumento de pesquisa, os respondentes são questionados quanto ao conhecimento, a utilização e o seu grau de formalismo. Os atributos pesquisados são:

- Suportabilidade
- Eficiência (Efficiency)
- Acurácia (Accuracy)
- Performance
- Robustez (Robustness)
- Conformidade (Correctness)
- Confiabilidade (Reliability)
- Integridade (Integrity)
- Usabilidade (Usability)
- Manutenabilidade (Maintainability)
- Testabilidade (Testability)
- Flexibilidade (Flexibility)

- Portabilidade (Portability)
- Reusabilidade (Reusability)
- Interoperabilidade (Interoperability)
- Segurança (Security)

O instrumento de pesquisa se encontra na íntegra no Apêndice 1 deste trabalho.

3.3.2 Fontes de dados e de informações

Os respondentes foram profissionais da área de TI de empresas de diversos portes e ramos de atividade, estabelecidas no Brasil, com contato com a área de desenvolvimento de *software*.

3.4 Procedimentos para acesso às empresas e coleta de dados

O dados foram coletados em sua maioria através da publicação de um instrumento em uma página na internet utilizando-se a ferramenta Omni Data Collector, com uma série de validações de consistência e de segurança de forma eletrônica, bem como a gravação das respostas em banco de dados. Os respondentes foram convidados a participar da pesquisa após uma seleção prévia do perfil de cada um, através de uma carta de apresentação contendo um endereço eletrônico para acesso ou um e-mail convite contendo o link. Vale ressaltar que uma série de mecanismos foram criados, como por exemplo gravação do endereço IP do usuário, evitando que um mesmo usuário responda o questionário mais de uma vez, ou mesmo chaves de segurança.

3.5 Tratamento de dados

Uma vez coletados, os dados e as informações sofreram um tratamento a fim de possibilitar a consolidação, os cruzamentos e a utilização de procedimentos estatísticos para as análises. Para isto, foi realizada a exportação do banco de dados contendo as informações coletadas para uma planilha Excel. Vale a pena ressaltar que apenas os dados consistentes foram exportados, já que através da ferramenta, foi realizada uma análise de outliers, bem como

respostas incoerentes e não consistentes. A partir daí, uma série de macros, filtros e tabelas dinâmicas foram construídas para a geração de gráficos para análise. Foram realizados uma série de cruzamentos com os dados levantados, para permitir a observação da concentração de determinadas práticas e atividades por porte da empresa, ramo de atuação, processo de desenvolvimento de *software*, utilização de ferramentas de teste, objetivo do *software* desenvolvido, tamanho da área de TI, e etc.

Os dados da primeira parte do questionário foram cruzados entre si, bem como serviram como base para o cruzamento com as informações das outras duas partes do questionário. Para a primeira parte, foram gerados gráficos de pizza mostrando a distribuição percentual de cada resposta para cada questão (variáveis), bem como para cada cruzamento entre variáveis. Já na segunda parte, com o auxilio da ferramenta Omni Data Collector, uma pontuação foi gerada para cada *drivers* ou critério, permitindo a construção, através do excel, de um gráfico de radar, mostrando simultaneamente o status das empresas em cada uma das dimensões, permitindo uma avaliação visual fácil dos critérios mais fortes e mais fracos. Foram gerados gráficos de radar considerando-se cada um dos portes de empresa, bem como os diferentes ramos de atuação, processo de desenvolvimento e etc. Já na terceira parte, a análise foi análoga à da primeira parte, tanto em termos dos gráficos, ou seja, gráficos de pizza com as distribuições percentuais das respostas, como os cruzamentos entre as variáveis.

Todos os gráficos gerados na etapa de análise dos dados coletados se encontram disponíveis no Apêndice 2 deste trabalho.

3.6 Figura-síntese de pesquisa

A Figura 41 apresenta um diagrama representando uma síntese do modelo de pesquisa utilizada nesta dissertação.

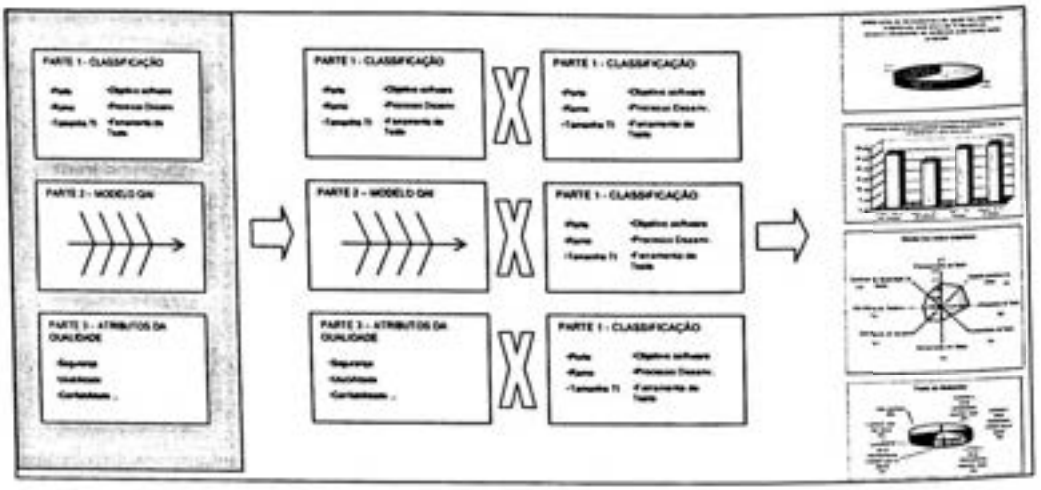

Figura 41 - Figura-síntese da Pesquisa

Basicamente o diagrama representa as três etapas da pesquisa. Inicialmente o instrumento de pesquisa contendo as três partes bem distintas tal como já descrito. Em seguida, após a coleta dos dados, foram realizados cruzamentos entre as variáveis da parte 1, entre as variáveis da parte 2 e da parte 1, e, por fim, entre as variáveis da parte 3 e da parte 1. Feitos estes cruzamentos, uma série de tabelas foram obtidas, possibilitando a geração dos diversos gráficos utilizados para a análise e mostrados no Apêndice 2 deste documento.

4 RESULTADOS

4.1 Introdução

Entre janeiro e março de 2005, o instrumento de pesquisa foi aplicado em profissionais da área de TI de empresas de pequeno, médio e grande porte, bem como micro empresas, todas estabelecidas no Brasil, com uma concentração maior daquelas presentes no Estado de São Paulo. Foi obtido um total de 112 questionários respondidos, utilizados para o processamento e análise. Tendo em vista este universo, esta seção visa apresentar os principais resultados obtidos a partir da análise dos dados obtidos nesta pesquisa.

4.2 Primeira Parte

No universo dos respondentes, a distribuição das empresas em relação ao porte (gráfico 1) mostrou a maior concentração em empresas de grande (45%) e pequeno porte (23%), enquanto as micro empresas representaram apenas 13% e as de médio porte, 19%. Considerando-se o setor de atuação das mesmas, o de maior predominância foi o de Tecnologia e Informática (42%), seguido pela Indústria e Comércio (14%), Prestação de Serviços (12%) e o Financeiro (8%). As empresas de telecomunicações e do setor financeiro são as que apresentaram, na entrevista, o maior contingente em sua área de TI, (66% e 100%, respectivamente, com mais de 100 pessoas).

De todo o universo, 92% das empresas realizam desenvolvimento de sistemas de informação (*software*), sendo 70% com muita freqüência e com objetivo, na maioria dos casos, destes *softwares* suportarem seus próprios negócios (50%) ou vendê-los como produto no mercado (30%). Quanto à subcontratação para esta atividade, ou mesmo a contratação de outras empresas, apenas 26% dos respondentes não a efetuam. Considerando-se o tamanho da área de TI destas empresas, 38% delas possuem mais de 100 pessoas, 24% de 30 a 100 pessoas e 25% até 10 pessoas, sendo que apenas 13% possuem de 10 a 30 pessoas. Os principais

compradores destes *softwares* são pessoas jurídicas (85%). A predominância das empresas entrevistadas que realizam desenvolvimento de *software*, com relação ao tamanho, foi a de grande porte (44%), seguida pela de pequeno porte (23%), e, com relação ao setor de atuação, o de Tecnologia e Informática (44%) e o de Prestação de Serviços (12%). Dentre as empresas que declararam não contratar ou subcontratar outras empresas para o desenvolvimento de *software*, a maioria (45%) é de pequeno porte.

Das empresas que desenvolvem sistemas de informação, a maior parte conduz estas atividades utilizando um processo em cascata (*waterfall*) (54%) enquanto 46% utilizam um processo em espiral. Considerando-se apenas as empresas de grande porte, o processo predominante é em cascata (69%), bem como nas micro empresas (56%). Nas de pequeno porte esta relação é invertida, já que a maioria apresenta um processo de desenvolvimento de *software* em espiral (31%). Apenas 38% das empresas médias entrevistadas desenvolvem *software* utilizando o processo em cascata. Esta modalidade de processo de desenvolvimento de *software* também é a mais praticada nas empresas atuantes no setor financeiro (83%) e na Indústria e no Comércio (70%), enquanto o processo em espiral é o mais praticado no setor de Prestação de Serviços (62%), Tecnologia e Informática (59%) e Telecomunicações (67%). Este ultimo é o processo que também predomina em empresas com uma área de TI com até 10 pessoas e com mais de 100 pessoas - em 59% e 52%, respectivamente.

A grande maioria (91%) das empresas afirma a existência de algum processo de validação ou teste para avaliação da qualidade do *software* desenvolvido, sendo que neste contexto, 87% dos entrevistados declararam que estas atividades sempre são realizadas pelos funcionários da empresa e apenas 2% delas sempre utilizam a mão-de-obra de uma empresa especializada no assunto. Considerando-se o porte das empresas entrevistadas, as de pequeno porte e as micro empresas foram as com a maior declaração negativa em relação à realização de tais validações/testes (15% e 22% das respostas foram negativas, respectivamente). Considerando-se o setor, 15% das empresas de Tecnologia e Informática declararam não realizar tais validações, bem como 13% das empresas de Prestação de Serviços. Na totalidade, as empresas entrevistadas de Indústria e Comércio, Telecomunicações e Financeira, afirmam possuir tais procedimentos. Dentre as empresas que não possuem tais validações, a maior parte utiliza o processo em cascata (11% frente aos 6% em espiral que não realizam validações/testes). Considerando-se o tamanho da área de TI, nas empresas com até 10 pessoas e naquelas com 30 a 100 pessoas, 18% e 13%, respectivamente, não realizam nenhum

tipo de validação ou testes para garantir a qualidade do *software* desenvolvido, representando as maiores concentrações na ausência de tais atividades. Em sua totalidade (100%), as empresas com foco em adaptação de pacotes de *software* e implantação em clientes realizam verificações ou testes de *software*.

Quanto à distribuição dos testes ao longo do desenvolvimento de *software*, no universo das empresas com processo de desenvolvimento de *software* em espiral e que realizam algum tipo de teste, um total de 73% delas sempre realiza estes testes ao final de cada iteração, enquanto 17% às vezes os executam neste momento e 3% somente ao final da última iteração.
Apenas 32% das empresas pesquisadas utilizam o suporte de ferramentas de teste de *software*. No entanto, dentre as demais, 74% pretendem utilizá-las, em sua grande maioria, em até 10 meses (71%). Os principais motivos apontados em não utilizar tais ferramentas foram o preço elevado (26%), a falta de conhecimento (10%), a precariedade da tecnologia (10%), o fato das mesmas agregarem pouco valor (8%) e o nível de maturidade insuficiente apresentado pela empresa (8%). As empresas que pretendem adotá-las, apontaram as ferramentas de automação de testes funcionais (32%), como a de maior intenção, seguida pelas ferramentas de testes de carga e estresse (23%). A ferramenta com uma menor intenção de uso, segundo os respondentes, é a de perfil de execução (5%). Levando-se em conta o tamanho, aquelas onde a utilização de ferramentas de apoio aos testes é mais presente são a de grande porte (35%) e de médio porte (33%). Em relação ao setor de atuação, o Financeiro (67%), Saúde (50%) e Tecnologia e Informática (39%), são aqueles em que a utilização de tais ferramentas se faz mais presente. Sob o prisma do processo de desenvolvimento de *software* utilizado, dentre aquelas que utilizam o modelo em espiral, 43% utilizam alguma ferramenta de testes, frente aos 22% no universo das empresas que utilizam o modelo em cascata. Sob a ótica do tamanho da área de TI, as empresas com mais de 100 pessoas utilizam estas ferramentas em sua maioria (56%). As empresas no setor de Telecomunicações, Indústria e Comércio, bem como Tecnologia e Informática, apresentam as maiores intenções de adoção de uma ferramenta de apoio aos testes – 100%, 86% e 86%, respectivamente. As empresas com foco no desenvolvimento de *software* para venda como um produto final são as que mais utilizam ferramentas de suporte aos testes (39%), seguida pelas empresas que desenvolvem soluções em *software* para suportar seus próprios negócios (31%). As ferramentas de testes mais utilizadas pelas empresas são a de automação de testes funcionais (23%) e a de teste de carga e estresse (27%). As ferramentas de perfil de execução (9%), perfil de memória consumida (9%), cobertura (9%), e obtenção de métricas de qualidade (9%) são utilizadas pelas empresas

entrevistadas em uma proporção equivalente, enquanto as ferramentas de análise estática de código fonte são utilizadas por 14% das empresas que utilizam ferramentas de apoio aos testes. No setor financeiro, as ferramentas de teste de carga e stress, automação de testes funcionais e cobertura são utilizadas em 28%, 28% e 18%, respectivamente, dos entrevistados que empregam tais recursos. Na Indústria e Comércio, os dois primeiros tipos de ferramenta são utilizados, ambas, por 40% dos entrevistados na mesma categoria. No setor de Tecnologia e Informática, estes números são 28% e 24% respectivamente. As empresas com processo em cascata que empregam ferramentas para testes, se concentram na utilização destas para teste de carga e estresse em 46% dos casos e para automação de testes funcionais e análise do perfil de memória em 18% em ambos. Já em se considerando as empresas com processo em espiral que empregam tais ferramentas, estes números são, respectivamente, 22%, 26% e 7%.

As empresas com foco em adaptação e implantação de pacotes de *software* em clientes são as que menos utilizam as ferramentas de teste de carga e stress (18%) e automação de testes funcionais (19%). Considerando-se os demais focos, tem-se 31% e 23% respectivamente, para desenvolvimento de *software* para venda como produto final e desenvolvimento de *software* para suporte aos próprios negócios.

Quando questionados sobre o nível de satisfação dos usuários dos *softwares* desenvolvidos, na média, os respondentes apontaram, dentro de sua percepção, em sua maioria níveis elevados (48% alto e 10% muito alto), enquanto apenas 3% acreditam que a satisfação seja baixa, sendo que esta insatisfação se concentra nas empresas de grande porte (nível de satisfação baixo na opinião de 6% dos respondentes de empresas grandes) e no setor financeiro (nível de satisfação baixo na opinião de 33% dos respondentes do setor financeiro). Aliás, os respondentes do setor financeiro foram os que apontaram os menores níveis de satisfação dos usuários, dentro de suas opiniões: 67% dos usuários com satisfação média e 33% dos usuários com satisfação baixa. Ou seja, nenhum respondente acredita que os usuários dos *softwares* desenvolvidos no setor financeiro possuam um nível de satisfação acima do mediano. No setor de prestação de serviços, a grande maioria dos respondentes apontou que em sua visão a satisfação dos seus usuários é mediana (62%). As empresas em que os usuários apresentam um nível baixo de satisfação na opinião dos respondentes, não utilizam ferramentas para testes de *software* (5%), bem como em sua totalidade, apresentam um processo de desenvolvimento em cascata (6%) e desenvolvem *software* para suportar seus próprios negócios (6%).

Os respondentes também apontaram, em sua visão, e em uma escala de 0 a 10, a percepção média dos usuários em relação à qualidade dos *softwares* desenvolvidos que eles utilizam. A média geral apresentada foi de 6,91. As empresas com foco em desenvolvimento de *software* para a comercialização como produtos finais, foram aquelas que apontaram as maiores notas, resultando numa maior média, 7,45, frente aos 6,62 e 6,70 das empresas com foco em implantação de pacotes de *software*, e empresas com desenvolvimento de *software* para suporte aos seus próprios negócios, respectivamente. Esta média também foi maior em empresas com desenvolvimento em espiral (7,26), em relação ao desenvolvimento em cascata (6,61). As empresas que utilizam ferramentas de testes também apresentaram uma média superior àquelas que não as utilizam (7,53 versus 6,54, respectivamente). Levando-se em conta o tamanho da área de TI, a categoria de 30 a 100 pessoas apresentou uma média 7,31, superior às demais categorias. Levando-se em conta o tamanho das empresas, as maiores médias ficaram para as empresas de médio e pequeno porte – ambas com 7,46. Tendo em vista o setor, as maiores médias foram na Indústria e Comércio (7,60), Energia Elétrica (7,50) e Saúde (7,50), ficando o setor Financeiro com a pior média – 4,50.

De forma análoga, os respondentes foram questionados quanto à sua percepção de qualidade em relação aos *softwares* desenvolvidos. Os resultados foram semelhantes, ficando a média geral em 6,93. Nas empresas que utilizam ferramenta de testes, a média também foi superior (7,68 frente aos 6,59 daquelas que não utilizam). A maior média, considerando-se o tamanho da área de TI, foi apresentada novamente pela categoria de 30 a 100 pessoas (7,44). As empresas de pequeno e médio porte apresentaram as melhores médias dentro da segmentação por porte da empresa (7,77 e 7,08, respectivamente). Esta análise dentro de uma segmentação por setor de atuação, apresentou as maiores médias para o setor de Energia Elétrica (8,00), Agroindústria (7,50) e Indústria e Comércio (7,40), ficando mais uma vez o setor Financeiro com a menor média (6,00). As empresas com desenvolvimento de *software* em espiral apresentaram mais uma vez uma média superior àquelas com desenvolvimento em cascata (7,26 e 6,64, respectivamente). Tendo em vista o objetivo do desenvolvimento de *software* das empresas entrevistadas, a média mais elevada (7,25) ficou para a categoria com foco em venda de soluções em *software* como um produto final, seguida da categoria com foco em suporte aos próprios negócios (6,91) e a adaptação e implantação de pacotes de *software* (6,62).

Os respondentes também apontaram a fração média do tempo total para o desenvolvimento de um *software*, dedicada às atividades de verificação e testes. A média geral do tempo dedicado a estas atividades, em relação ao tempo total, foi de 25,44%. Segmentando-se pelo setor de atuação, os maiores tempos médios foram apresentados pelo setor de Energia Elétrica (27,50%), Prestação de Serviços (26,88%) e Tecnologia e Informática (26,22%), enquanto a Agroindústria (10,00%), o setor Financeiro (15,00%) e o Governo (18,00%) apresentaram os menores tempos médios dedicados às atividades de verificação e testes de *software*. Levando-se em conta o tamanho da área de TI, aquelas com até 10 pessoas apresentam a maior média (30,29%), enquanto as menores são daquelas com mais de 100 pessoas (23,72%) e com entre 10 e 30 pessoas (23,33%). Uma segmentação de acordo com o porte da empresa apontou os maiores tempos para as empresas pequenas (28,08%) e os menores para as empresas médias (21,54%) e grandes (25,56%). As empresas que empregam ferramentas em seus testes possuem as menores frações de tempo dedicada aos testes (24,21%) em relação àquelas que não utilizam o suporte de ferramentas nas atividades de testes (26,10%).

4.3 Segunda Parte

Na segunda parte os respondentes foram questionados para a obtenção de uma série de dados com o intuito de aplicar o modelo de avaliação de processos de teste criado pelo QAI (Quality Assurance Institute) e descrito por William E. Perry em sua obra *Effective Methods for Software Testing*, para a avaliação das empresas no que tange a presença de disciplinas e práticas com o intuito de garantir a qualidade dos *softwares* desenvolvidos. Para cada questionamento respondido afirmativamente, foi atribuída uma pontuação, enquanto que para cada questionamento respondido negativamente, não foi atribuída nenhuma pontuação. Como cada uma das "dimensões" avaliadas pelo modelo é composta por um conjunto das questões aplicadas, para cada uma delas, foi calculado a pontuação total somando-se os pontos obtidos ou não em cada um dos questionamentos pertencentes a esta dimensão. A pontuação mínima para cada dimensão é 0 enquanto que a máxima é 5. Desta forma, o valor da pontuação de cada pergunta é calculado como uma fração, onde se todas as perguntas de uma dimensão forem respondidas afirmativamente, a soma para esta totalize 5. Como para cada dimensão o número de perguntas aplicadas é variável, o valor de pontos possíveis de serem acumulados para cada pergunta varia de acordo com cada dimensão.

Para a apresentação dos resultados e análise, o autor sugere a construção de um gráfico de radar, tal como foi feito e é apresentado no gráfico 159. Desta forma, pode-se analisar de forma visual as dimensões mais e menos desenvolvidas.

Este gráfico apresenta uma avaliação média de todas as empresas entrevistadas de acordo com o modelo utilizado. Neste contexto, a dimensão mais desenvolvida é a de Planejamento do Teste, seguida pela de Processos de Teste, enquanto um desenvolvimento precário é encontrado no Treinamento em Teste e Ferramentas de Teste.

Analisando-se separadamente esta fotografia em empresas de grande, médio e pequeno porte (gráficos 160, 161, 162), estas tendências se mantém com alguma variação. Já para as micro empresas (gráficos 163), as polarizações são menores, havendo um maior equilíbrio entre as dimensões, às custas de um menor Planejamento no Teste e um maior Suporte Gerencial ao Teste.

Realizando esta avaliação dentro de cada setor de atuação, o da Indústria e Comércio (gráfico 167) foi o que apresentou os menores índices para a Eficiência do Teste, Treinamento em Teste e Ferramentas de Teste, bem como o segundo menor índice para o Controle de Qualidade do Teste. O menor índice de Controle de Qualidade do Teste foi o apresentado pelo setor de Energia Elétrica (gráfico 164), que também apresentou um índice para a Eficiência do Teste comparável à Indústria e Comércio. Por outro lado, ambos os setores apresentaram os mais altos índices no Planejamento do Teste. Os setores com os índices mais altos para os Processos de Teste foram o de Saúde (gráfico 169) e o de Telecomunicações (gráfico 171), respectivamente. Os setores de Energia Elétrica e Saúde foram os que apresentaram os índices mais altos no Suporte Gerencial ao Teste, nesta ordem. O menor índice de Satisfação do Usuário, de acordo com este modelo, ocorreu no setor Governamental (gráfico 166).

Aplicando-se o modelo de avaliação separadamente para empresas com desenvolvimento de *software* em cascata e em espiral (gráficos 172 e 173, respectivamente), observa-se menores polarizações, ou seja, uma distribuição mais harmônica, pelas dimensões, nas empresas que adotam o modelo em espiral. Nestas, apenas a dimensão Satisfação do Usuário, apresentou um índice inferior, enquanto todas as outras dimensões foram superiores, em comparação à média das empresas com desenvolvimento em cascata.

O mesmo modelo aplicado aos três diferentes objetivos do desenvolvimento de *software*, apontou que as empresas que visam a comercialização do *software* como um produto final (gráfico 176) possuem os índices mais elevados em Treinamento em Teste, Ferramentas de Teste, Processos de Teste, Suporte Gerencial aos Teste e Planejamento do Teste, bem como um segundo menor índice na dimensão Satisfação do Usuário. Nesta dimensão, o maior índice é das empresas com foco no desenvolvimento para o suporte aos próprios negócios (gráfico 175). Estas possuem ainda os maiores índices para as demais dimensões: Controle de Qualidade do Teste e Eficiência do Teste. As empresas com foco em adaptação e implantação de pacotes de *software* em clientes (gráfico 174) possuem os menores índices para o Planejamento do Teste, o Suporte Gerencial ao Teste, os Processos de Teste, as Ferramentas de Teste, o Treinamento em Teste, a Satisfação do Usuário e a Eficiência do Teste, bem como o segundo menor índice para a dimensão restante: Controle de Qualidade do Teste.

4.4 Terceira Parte

Na ultima parte do instrumento de pesquisa, os respondentes forneceram informação quanto ao grau de conhecimento de alguns atributos de qualidade de *software* e a prática de testes para avalia-los.

Os atributos menos conhecidos são a Acurácia (40%), Suportabilidade (40%), Interoperabilidade (35%), Testabilidade (30%) e Robustez (29%). Os atributos Confiabilidade (24%), Conformidade (32%), Integridade (32%), Manutenabilidade (25%), Performance (26%), Segurança (35%), Testabilidade (24%) e Usabilidade (36%) são os que apresentam os maiores índices de respostas do tipo "Conhece e utiliza formalmente sempre que se aplicar", com relação ao conhecimento do atributo por parte do respondente e a utilização de testes para a validação deste.

Com relação ao atributo Acurácia, o índice de desconhecimento supera a média de 40% considerando-se as empresas de grande porte (42%), ou as empresas no setor Financeiro (50%) e Telecomunicações (50%), ou ainda as empresas com desenvolvimento em cascata (49%).

O desconhecimento do atributo Suportabilidade foi acima da média de 40% considerando-se as empresas da Indústria e Comercio (50%), Telecomunicações (50%) e Tecnologia e Informática (42%), ou, as empresas com desenvolvimento de *software* em cascata (49%).
A mesma análise para o atributo Interoperabilidade, mostra um desconhecimento acima da média (35%), considerando-se as empresas de grande porte (37%) e de médio porte (40%), ou as empresas do setor Financeiro (50%) e da Indústria e Comércio (60%), ou ainda, as empresas com desenvolvimento de *software* em cascata (38%).
Esta análise aplicada ao atributo Testabilidade revela um desconhecimento acima da média (30%) em se considerando as empresas de médio porte (50%), de pequeno porte (33%) e micro empresas (40%), ou, a Indústria e Comércio (60%), Tecnologia e Informática (50%).
Levando-se em consideração as empresas de pequeno e médio porte, ambas apresentaram um índice de desconhecimento acima da média para o atributo Robustez (33% e 40%, respectivamente, em relação a uma média de desconhecimento de 29%). Analisando-se o setor da empresa, o Financeiro, a Indústria e Comércio e Tecnologia e Informática, apresentam índices de desconhecimento de 50%, 40%, e 42%, respectivamente. Com relação ao processo, nas empresas com desenvolvimento em cascata, o desconhecimento do atributo possui uma média de 44%.

Os respondentes das micro-empresas entrevistadas em 50% dos casos declaram conhecer o atributo Confiabilidade e utilizar testes para avaliá-lo sempre que se aplicar (acima da média de 24%). As empresas do setor de Prestação de Serviços, de Tecnologia e Informática e de Telecomunicações são as que mais conhecem o atributo e aplicam os testes para avaliá-lo nos *softwares* nas situações cabíveis (25%, 42% e 25%, respectivamente).

Levando-se em conta o atributo Conformidade, as empresas de grande e pequeno porte, bem como as micro-empresas possuem uma média de conhecimento e utilização, sempre que se aplicar, igual ou superior à média (32%, 50% e 33%, respectivamente, diante da média de 32%). Todos os respondentes das empresas do setor de Telecomunicações declararam conhecer e utilizar testes para validar este atributo sempre que for aplicável, enquanto que ste índice é de 50% em empresas de Tecnologia e Informática.

O atributo Integridade é conhecido e testes para sua avaliação são aplicados sempre que forem cabíveis para 32% dos entrevistados. Este índice é acima da média apresentada (32%) nas empresas de pequeno (33%) e médio porte (60%), bem como considerando-se o setor de

Prestação de serviços (50%), Tecnologia e Informática (42%) e as empresas com desenvolvimento em espiral (38%).

A Manutenabilidade apresentou o índice de 25% para a mesma categoria de respostas, sendo que acima desta média encontram-se as empresas de médio (30%) porte e micro empresas (50%), bem como as empresas de Tecnologia e Informática (27%) e Telecomunicações (50%). Este mesmo índice médio foi de 26% para o atributo Performance, em se considerando o mesmo tipo de resposta (Conhece e utiliza formalmente sempre que se aplicar). Acima desta média se encontram as empresas de grande (27%) e médio (30%) porte, bem como as micro-empresas (40%) entrevistadas e as empresas de Tecnologia e Informática (33%). Já para o atributo Segurança, com média de 35% para o mesmo tipo de resposta, os índices acima deste valor são apresentados pelas empresas de grande (36%) e médio (40%) porte, assim como as micro-empresas (40%). Considerando-se o setor, o Financeiro apresenta uma média de 50% para este tipo de resposta, assim como Tecnologia e Informática um valor de 34% e Telecomunicações um índice de 75%.

Em se tratando do atributo Testabilidade, para o mesmo tipo de resposta, o índice foi de 24%, sendo superado pelas empresas de grande (26%) e médio porte (30%), bem como pelas empresas no setor Financeiro (50%) e Telecomunicações (50%). O atributo Usabilidade apresentou uma média de 36% para este mesmo tipo de resposta, sendo superado pela média das empresas de médio (40%) porte e micro-empresas (40%), bem como empresas do setor Financeiro (50%), da Indústria e do Comércio (40%) e de Prestação de Serviços (50%).

5 CONCLUSÕES

5.1 Introdução

Este capítulo apresenta as conclusões deste trabalho no que tange à análise dos dados coletados e processados da pesquisa realizada. Além disto, neste tópico, algumas sugestões de futuros estudos, bem como algumas das limitações presentes neste são apresentadas.

5.2 Conclusões a partir da revisão bibliográfica

Olhando para o desenvolvimento de *software*, de uma perspectiva histórica, tal como é descrito por Kan (2003, p. xix-xxi) os anos 60 podem ser vistos como a era funcional, os anos 70 como a era do cronograma, os anos 80 como a era do custo, e os anos 90 até os dias atuais como a era da qualidade e da eficiência. No anos 60, as empresas aprenderam como explorar a tecnologia da informação para atender as necessidades institucionais bem como integrar o *software* às operações diárias das empresas. Nos anos 70, a indústria foi caracterizada pelos atrasos maciços de cronograma, bem como estouros de orçamentos, trazendo o foco para o planejamento e o controle destes projetos. Os modelos de ciclo de vida do *software* baseados em fases foram introduzidos, bem como técnicas de análises. Nos anos 80 os custos do hardware continuaram a cair, e a tecnologia da informação permeou por cada área das empresas, tornando-se disponível para os indivíduos. Conforme a competição na indústria se tornou acirrada e as aplicações de baixo custo foram amplamente implementadas, a importância da produtividade no desenvolvimento do *software* aumentou significativamente. Nesta época vários modelos de custos foram desenvolvidos e utilizados. No final desta década, a importância da qualidade do *software* passou também a ser reconhecida. (KAN, 2003)

A partir de então, pode-se considerar a era da qualidade do *software*. Com o estado da arte da tecnologia disponível para prover funcionalidades em abundância, os clientes demandam alta qualidade. A demanda pela qualidade é também intensificada pela dependência cada vez

maior da sociedade em cima de *softwares*. Ou seja, há 50 anos, uma central telefônica ou um míssel iriam falhar provavelmente por um problema eletro-mecânico. Hoje a qualidade nestas áreas do conhecimento provê confiabilidade e uma tolerância a falhas suficiente para satisfazer as nossas necessidades, no entanto, a capacidade de atender as necessidades de flexibilidade, usabilidade e, principalmente, funcionalidade, são restritas, de forma que o *software* vem preencher esta lacuna. Entretanto, a qualidade do *software* não atingiu um nível equivalente ao destas tecnologias, de forma que se por um lado, as necessidades fazem com que cada vez mais este seja empregado, por outro, estamos cada vez mais aumentando o risco de falha em equipamentos que já tinham atingido um patamar de confiabilidade e estabilidade desejáveis.

Por outro lado, a criação de interfaces visuais para as aplicações, aumentou a usabilidade e a acessibilidade a um público não técnico, de forma que, se há 40 anos o *software* era utilizado por quem o escrevia, hoje ele é utilizado por uma gama diferente de pessoas que não necessariamente o utilizarão da forma como ele foi projetado, seja pela falta de know-how ou por uma necessidade não tão congruente com as suas funcionalidades. Os resultados são o exercício de cenários de utilização provavelmente não imaginados e que potencializam a exposição de erros.

O advento e a massificação das tecnologias de rede de computadores, principalmente a internet, possibilitou o atendimento de uma enorme gama de serviços, bem como a quebra de alguns paradigmas e a criação de alguns novos modelos de negócio. Hoje a grande maioria das transações financeiras são realizadas de forma eletrônica, possibilitando-se movimentar valores por diversas regiões do mundo apenas pelo tráfego de informações por redes digitais de computadores. Por outro lado, tecnologias como a internet trouxeram a preocupação com dois temas: a performance e a segurança. A performance passou a ser crítica, pois antes um *software* era utilizado por um usuário e em um computador, com exceção dos computadores de grande porte que permitiam o compartilhamento de recursos. Agora, os *softwares* em um servidor são acessados por um número grande de usuários e de uma forma não controlável, fazendo com que a disputa pelos recursos de tráfego na rede e processamento nos servidores tragam impacto nos tempos de resposta das aplicações. Outro ponto é o acesso potencial que qualquer usuário no mundo tem a qualquer rede de qualquer tipo de empresa. Informações que deveriam ser confidenciais e são tratadas como tal, podem facilmente ser acessadas explorando-se as vulnerabilidades destas redes, bem como dos sistemas operacionais,

aplicativos para navegar na internet, ou mesmo para leitura de e-mail. Se por um lado a internet e a tecnologia da informação trouxeram o paraíso em termos de suporte a processamentos volumosos e disponibilidade de informações de uma forma inteligente para suportar a tomada de decisões, por outro, trouxeram o inferno no que tange a segurança das informações, ou a confiabilidade, integridade, autenticidade e não repúdio de informações.

Nitidamente, a indústria de *software* se movimenta em passos cada vez menos tímidos para a busca de uma maior qualidade nestes produtos, seja pela maior exigência dos consumidores, seja por medidas judiciais cada vez mais presentes em países da Europa e nos Estados Unidos. Uma primeira dificuldade que surge no tema garantia da qualidade de *software* é a definição de qualidade de *software* e os itens que a compõe. Existem inúmeras definições, sendo que as mesmas seguiram uma evolução ao longo do tempo de uma visão advinda da indústria tradicional, para o atendimento das necessidades do consumidor de *software*, dentro de uma série de atributos ou dimensões definidas por diferentes modelos de qualidade. Estes atributos variam de acordo com cada modelo, seja o de McCall, o FURPS ou mesmo a ISO/IEC 9126, dentre outros, mesmo havendo algum denominador comum. Esta variação é natural, pois se de um lado a preocupação na década de 60, por exemplo, era principalmente com a funcionalidade, 35 anos depois, com uma dependência maior de sistemas de informações e com o surgimento da internet, esta passou a ser mais uma dentro de outras tantas, como performance, confiabilidade, segurança, etc.

Além da ausência de unanimidade em uma definição de qualidade, e em um modelo, uma outra dificuldade é como mensurar cada um dos atributos da qualidade do *software*. Uma das formas é a realização de testes. No entanto, existem mais de 200 técnicas de testes, com uma ampla variação no grau de simplicidade e eficiência. Algumas delas podem ser realizadas por ferramentas, outras não, sendo que em se tratando de ferramentas, existe uma enorme variedade de tipos e aplicações, bem como valores, com cenários muito limitados, em alguns casos, onde os resultados podem agregar valor. Por outro lado, para realizar o teste funcional completo, envolvendo todas as possibilidades de uma *software* contendo 5 telas, por exemplo, pode-se levar centenas de milhões de anos. Adicionando-se testes de performance, segurança, e aqueles relacionados a outros atributos da qualidade, provavelmente este tempo será um pouco maior. Ou seja, se o teste completo é impossível, se a presença de *bugs* latentes em aplicações é tida quase que como um axioma, e como o teste de aplicações demanda recursos humanos, tempo e ferramentas, torna-se uma tarefa complicada defender os testes de

aplicações em certos cenários, ou pelo menos estabelecer regras claras e uma região operacional ótima onde exista a melhor relação custo benefício. Trata-se de um grande desafio a ser enfrentado em algumas frentes - evolução dos conceitos de qualidade de projeto bem como sua aplicação cada vez mais presente; evolução dos conceitos de qualidade de processo bem como sua utilização; evolução dos conceitos de qualidade de *software* bem como sua utilização, evolução de ferramentas de apoio aos testes para realizar de forma inteligente e rápida, testes complexos e que potencialmente durariam mais do que a expectativa de vida de um ser humano. Sem dúvida alguma, um importante trabalho deve ser realizado: ajustar as expectativas dos gestores de projetos que o teste não elimina 100% dos erros, apenas reduz o risco contido em um *software*. Enquanto esta expectativa perdurar, o teste continuará a ser uma tarefa pouco valorizada dentro do desenvolvimento de *software*, já que trata-se de um investimento incapaz de atingir o objetivo de eliminar todos os erros das aplicações, dentro da visão presente em muitos casos.

5.3 Conclusões a partir da análise da pesquisa

Talvez o ponto mais importante a ser levantado neste item, inicialmente, é um grande viés da pesquisa realizada. As empresas onde os respondentes trabalham pertencem a um universo de empresas de relacionamento do autor, justamente pelo motivo deste atuar em consultorias em qualidade e teste de *software*. Desta forma, estas empresas são potencialmente mais propensas a utilizar diferentes técnicas de teste, ferramentas de apoio aos testes, e outras disciplinas da qualidade de *software*. Ou, em outras palavras, as empresas entrevistadas, potencialmente estariam alguns passos à frente das demais em termos de qualidade de *software* e testes de *software*. No entanto, se por um lado esta informação pode depreciar a pesquisa, por outro ela valoriza este trabalho se for considerado que os resultados demonstraram que as empresas estão muito aquém do que deveriam – classe mundial dentro dos conceitos do QAI (Quality Assurance Institute) - em termos de processo de testes (parte II do instrumento de pesquisa) e conhecimento dos atributos da qualidade de *software* e utilização de testes para avaliação dos mesmos (parte III do instrumento de pesquisa). Provavelmente, se esta pesquisa fosse feita em uma base maior de empresas não se restringindo ao universo de conhecimento do autor, os resultados seriam ainda piores. Desta forma, um estudo com esta amplitude é uma primeira sugestão de continuidade desta pesquisa.

Um dos pontos que chama atenção nos resultados desta pesquisa é o percentual baixo de tempo médio dedicado aos testes de *software*, principalmente em alguns setores de negócio, comparando-se ao que a literatura especializada recomenda, tal como descrito na revisão bibliográfica. O setor financeiro, um dos que mais dependem de tecnologia da informação para viabilizar suas operações, é o que menos investe tempo em testes (15% do tempo de desenvolvimento frente a uma média de 25,44%). Este é um setor que possui um risco elevadíssimo associado às falhas de *software*, pois os prejuízos financeiros estão quase que diretamente relacionados às falhas e é uma das áreas, segundo o NIST (Instituto Nacional de Normas e Tecnologias dos Estados Unidos), que pode economizar mais com a adoção de uma infra-estrutura adequada de testes. Se por um lado este baixo tempo poderia ser devido à utilização de ferramentas de apoio aos testes, que muitas vezes automatizam as atividades, economizando tempo, dado que o setor financeiro possui a maior fração de utilização destas, por outro este é o setor em que os respondentes apontaram as notas mais baixas em termos de percepção de qualidade dos *softwares*, bem como em termos de níveis de satisfação. Desta forma, mesmo que o tempo baixo seja justificável pela automação de atividades de teste, a efetividade dos testes pode ser considerada baixa.

A pesquisa mostrou que um grande número de empresas adota o processo iterativo em espiral para o desenvolvimento de *software* (46%). Neste universo, quando comparado com o universo das empresas que desenvolvem *software* em cascata, as avaliações em termos de percepção de qualidade do respondente são superiores.

Dentre as empresas entrevistadas, apenas 2% daquelas que realizam testes de aplicações, declararam utilizar mão-de-obra de empresas especializadas para a realização destas validações. Este é um número baixo em se tratando de uma atividade especializada e que requer conhecimentos muito profundos e específicos, bem como experiência. Além disto a utilização de mão de obra de uma empresa especializada evita a utilização de recursos que atuam no desenvolvimento do *software*, evitando graves vieses nos testes, principalmente no que tange a utilização de cenários para os quais ele foi projetado. A utilização de empresas especializadas é uma novidade no mercado e é vista como uma tendência, para o aumento da eficiência e confiabilidade dos testes, bem como uma alternativa a ser trabalhada dentro da terceirização do desenvolvimento. Neste cenário, a empresa passa todo o desenvolvimento de *software* para uma fábrica de *software* ou empresa especializada em desenvolvimento, para

que esta atenda suas necessidades em termos de sistemas de informação para suporte aos seus negócios. No entanto, as experiências norte-americanas com a Índia, por exemplo, mostraram que um número elevado de contratos é rompido de forma litigiosa, e, que a qualidade do *software* desenvolvido diminui acentuadamente. Neste contexto de terceirização, uma empresa com foco em testes de *software* pode aferir a qualidade do produto sendo gerado pela empresa produtora, bem como avaliar se o mesmo atende corretamente às necessidades da empresa cliente.

A segunda parte da pesquisa permitiu um levantamento resultando em uma visão do quão distante as empresas se encontram em termos de processo de teste de uma organização de teste de *software* de classe mundial. Percebe-se que apenas o atributo Planejamento de Testes está próximo do que é considerado classe mundial no modelo utilizado (QAI). O itens mais distantes do "ideal" do modelo são o Treinamento em Testes e a utilização de Ferramentas de Testes.

A baixa utilização de ferramentas de testes deve-se, de acordo com o que foi apontado pela pesquisa, em primeiro lugar, ao seu preço ser considerado elevado pelos respondentes. Isto é plausível, já que além do fato delas realmente possuírem um elevado custo em termos absolutos, já que são produzidas fora do Brasil mantendo-se a política de preços destes países aqui, são caras também do ponto de vista relativo ou do retorno sobre o investimento. Em países como os Estados Unidos, o retorno sobre o investimento nestas ferramentas pode ser facilmente alcançado calculando-se o valor do trabalho que ela realiza se fosse realizado por um funcionário. Já aqui, tem-se o mesmo valor da ferramenta com um valor muito inferior da mão de obra, de forma que o retorno sobre o investimento ocorre em um horizonte pouco atraente.

A terceira parte da pesquisa mostrou que vários atributos são pouco conhecidos, e, portanto, as empresas que não os conhecem não realizam testes para avaliá-los. Dentre os atributos da qualidade de *software* menos conhecidos estão: Acurácia, Suportabilidade, Interoperabilidade, Testabilidade e Robustez. Já na esfera dos atributos mais conhecidos, e, avaliados formalmente em situações em que se aplicam, se encontram a Confiabilidade, Conformidade, Integridade, Manutenabilidade, Performance, Segurança, Testabilidade e Usabilidade.

Como sugestão de continuidade deste estudo, poder-se-ia realizar anualmente a aplicação do mesmo instrumento no mesmo universo de empresas entrevistadas, para avaliar o grau de evolução em direção à um processo de testes de classe mundial, o aumento do conhecimento e avaliação dos atributos de qualidade, bem como relaciona isto à percepção de qualidade dos desenvolvedores.

Outra limitação desta pesquisa que pode originar uma continuidade deste estudo é em cada uma das empresas em que este instrumento foi aplicado, aplicar um instrumento para avaliar a percepção de qualidade dos usuários e verificar a correlação existente entre o grau de evolução dos processos de teste, tempo dedicado ao teste, utilização de ferramentas de teste, e a realização de avaliações dos atributos com a satisfação dos usuários. Outra medida interessante seria a análise do *gap* existente entre o grau de satisfação do usuário em relação ao *software* desenvolvido na percepção do recurso humano ligado à área de TI, tal como foi realizado na presente pesquisa, e o grau de satisfação do usuário do *software* em sua visão (coletado diretamente com o usuário).

REFERÊNCIAS

BARTIÉ, Alexandre. **Garantia da Qualidade de Software**: As melhores práticas de engenharia de software aplicadas à sua empresa. Rio de Janeiro: Ed. Campus, 2002.

BEIZER, Boris. **Software Testing Techniques**. 2nd ed. Scotsdalle: The Coriolis Group, 1990.

BROEKMAN, Bart; NOTENBOOM, Edwin. **Testing Embedded Software**. London: Pearson Education Limited, 2003.

BURNSTEIN, Ilene. **Practical Software Testing**: A process-oriented approach. New York: Springer-Verlag New York, Inc., 2003.

COPELAND, Lee. **A Practitioner's Guide to Software Test Design**. Boston: Artech House, 2004.

COSTA, Hudson R. **Gestão do Processo de Teste de Software:** Dentro da Metodologia Rational Unified Process (RUP). Brasília, 2003. Disponível em: <http://www.optimize.com.br/downloads/Trabalho_Teste_Hudson.pdf>. Acesso em: 20/03/2005.

CRAIG, Rick D.; JASKIEL, Stefan P. **Systematic Software Testing**. Norwood: Artech House, 2002.

CRSIP, Computer Resources Support Improvement Program. **Software Quality Assurance**: A technical report outlining representative publications on the subject. Revision 0.2. [S.l], April 6, 2000.

DE YOUNG, Clinton; RÄTZMANN, Manfred. **Software Testing and Internationalization**. Salt Lake: Lemoine International, Inc., 2003.

DUSTIN, Elfriede. **Effective Software Testing**: 50 Specific Ways to Improve Your Testing. Boston: Pearson Education, Inc., 2003.

EDGE, Rational. **The Spirit of the RUP**. Dezembro, 2001. Disponível em: <http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/dec01/TheSpiritoftheRUPDec01.pdf>. Acesso em: 06/05/2005.

FISHMAN, Charles. *They write the right stuff*. **Fast Company Magazine.** [S.l.], v. 6, 12/1996-01/1997. Disponível em: < http://www.fastcompany.com/magazine/06/writestuff.html >. Acesso em: 20/06/2004.

GRAHAM, Dorothy; FEWSTER, Mark. **Software Test Automation:** Effective use of test execution tools. London: Addison-Wesley, 1999.

HORCH, John W. Practical Guide To Software Quality Management. 2nd ed. Boston: Artech House, 2003.

HUMPHREY, Watts S. *Defective Software Works*. **What's New.** [S.l.], v. 1, 2004. Disponível em: <www.sei.cmu.edu/news-at-sei/columns/watts_new/2004/1/watts-new-2004-1.pdf>. Acesso em: 30/03/2005.

HUTCHESON, Marnie L. **Software Testing Fundamentals.** Indianopolis: Wiley Publiching Inc, 2003.

INFORMÁTICA. *77% das empresas no Brasil tiveram sistema de TI atacado em 2003.* **Folha OnLine.** São Paulo, 13/11/2003. Disponível em: <http://www1.folha.uol.com.br/folha/informatica/ult124u14449.shtml>. Acesso em: 28/11/2004.

INTHURN, Cândida. **Qualidade & Teste de Software**. São José, SC: Visual Books, 2001.

KAN, Stephen H. **Metrics and models in software quality engineering**. 2nd ed. Boston : Addison-Wesley - Pearson Education, Inc., 2003.

KANER, Cem *et al*. **Lessons learned in software testing:** A context-driven approach. New York: John Wiley & Sons, Inc., 2002.

KANER, Cem *et al*. **Testing Computer Software**. 2nd ed. New York: John Wiley & Sons, Inc., 1999.

KRUCHTEN, Philippe. **The Rational Unified Process**: An Introduction. Massachusetts: Addison Wesley, 1999.

LEWIS, William E. **Software testing and continuous quality improvement**. Boca Raton: CRC Press LLC, 2000.

LI, Kaglin; WU, Menqi. **Effective Software Test Automation**: Developing an Automated Software Testing Tool. San Francisco: Sybex Inc., 2004.

MCCONNELL, Steve. **Professional Software Development**: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers. Boston: Addison Wesley, 2003.

MYERS, Glenford J. **The Art of Software Testing**. 2nd ed. Hoboken: John Wiley & Sons, Inc., 2004.

PÉREZ, Maria *et al*. **A Systemic Quality Model for Evaluating Software Products.** Venezuela. Disponível em: < http://seir.sei.cmu.edu/seir/domains/FRMSET.PDF.ASP?ACTION=view&DOMAIN=AcquisitionRisk&SECTION=Implement&SUBSECTION=Documents&SUBSUBSECTION=Acqui

sitionManagement&DOC=CONTRIBUTION-5-13-2004&MEDIA=Paper&IMG=&MAX=0>. Acesso em: 20/03/2005.

PERRY, William E. **Effective methods for software testing**. 2nd ed. New York : John Wiley & Sons, Inc., 2000.

RATIONAL Unified Process. [S.l.], Rational Software Corporation, 2002. CD-ROM

REUTERS. **Software Errors Cost Billions**. Yahoo News, 28/06/2002. Disponível em: <http://story.news.yahoo.com/news?tmpl=story&cid=581&ncid=581&e=7&u=/nm/20020628/tc_nm/tech_software_dc_1>. Acesso em: 30/08/2004.

RIOS, Emerson; MOREIRA FILHO, Trayahú R. **Projeto & Engenharia de Software**: Teste de Software. Rio de Janeiro: Alta Books, 2003.

ROSS, Kevin. **Practical Guide to Software System Testing**. [S.l.] K. J. Ross & Associates Pty. Ltd., 1998.

SEGURANÇA Digital. **Brasil é líder mundial em invasões**. Batori Security. Novembro, ano 1, n. 23, 11/11/2003. Disponível em: <http://www.batori.com.br/boletim/edicoes/Numero0023.htm#artigo3>. Acesso em: 20/04/2004.

SPLAINE, Steven. **Testing Web Security**: Assessing the Security of Web Sites and Applications. Indianapolis: Wiley Publishing, Inc., 2002.

SYKES, David A.; MCGREGOR, John D. **A practical guide to testing object-oriented software**. Massachusetts: Addison-Wesley, 2001.

TSAO, H.-S. Jacob et al. **Testing and Quality Assurance for Component-Based Software**. Boston: Artech House, 2003.

WHITTAKER, James A. **How to Break Software**: A Practical Guide to Testing. Boston: Pearson Education, Inc., 2003.

APÊNDICES

APÊNDICE 1 – INSTRUMENTO DE PESQUISA APLICADO

Parte 1 – Classificação do respondente

Dados do Respondente

1-1) Qual o tamanho da empresa onde trabalha? (R.U.)

( ) Micro - 1 a 9 pessoas
( ) Pequena - De 10 a 99 pessoas
( ) Média - de 100 a 499 pessoas
( ) Grande - 500 ou mais pessoas

2-2) Qual é o ramo de atuação da empresa onde trabalha? (R.U.)

( ) Financeiro
( ) Governo
( ) Militar
( ) Indústria e Comércio
( ) Tecnologia / Informática
( ) Prestação de Serviços
( ) Telecomunicações
( ) Comércio / Varejo
( ) Energia Elétrica
( ) Educação
( ) Saúde
( ) Outros. Qual? _____

3-3) Qual é sua posição na empresa? (R.U.)

( ) Executivo - Diretor, Vice-Presidente ou Presidente
( ) Gerencial - Gerente, Coordenador
( ) Operacional - Analista, Consultor, Auxiliar, Programador, Testador, e outros

4-4) A empresa onde trabalha realiza desenvolvimento de Sistemas de Informação (Software)? (R.U.)

( ) Sim
( ) Não

5-5) A empresa em que trabalha sub-contrata ou contrata outras empresas para o desenvolvimento de software? (R.U.)

( ) Não
( ) Às vezes
( ) Sempre

Dados do Respondente / Classificação 1

6-As próximas perguntas somente deverão ser respondidas se você escolheu a alternativa "a" (SIM) na questão 4 e NÃO escolheu a alternativa "a" (NÃO) na questão 5.

6) Como é conduzido o processo de desenvolvimento de software? (R.U.)

( ) Em cascata, ou seja, inicialmente se coleta os requisitos, depois faz-se uma modelagem do sistema, em seguida a codificação do mesmo, e por fim, os testes antes de colocá-lo para utilização em produção.

( ) Em espiral, ou de forma iterativa, na qual o desenvolvimento é dividido em vários sub-projetos seqüenciais, onde em cada iteração, cada um destes sub-projetos é executado, com finalidade de desenvolver um software contendo uma parte do sistema total.

7-7) Com qual freqüência sua empresa realiza, participa ou contrata um projeto de desenvolvimento de software? (R.U.)

( ) Raramente
( ) Com pouca freqüência
( ) Com muita freqüência

8-8) Existe algum tipo de validação ou teste, que é realizado para avaliar a qualidade do software, seja ele comprado ou seja ele desenvolvido internamente? (R.U.)

( ) Sim
( ) Não (pular para 16)

9-9) Estas validações ou testes são realizados por uma empresa especializada ou por funcionários da empresa em você que trabalha? (R.U.)

( ) Sempre pelos funcionários da empresa
( ) Às vezes pelos funcionários da empresa, outras vezes, por uma empresa especializada
( ) Sempre por uma empresa especializada

10-10) Estas validações são realizadas ao final de cada iteração de desenvolvimento? (Somente responda esta pergunta se na questão 6 a sua resposta foi a alternativa "Em espiral") (R.U.)

( ) Sempre
( ) Às vezes
( ) Somente ao final de algumas iterações
( ) Somente ao final da ultima iteração
( ) Nunca

11-11) Você realiza testes com a ajuda de alguma ferramenta? (R.U.)

( ) Sim
( ) Não

12-12) Qual tipo de ferramenta? (Somente responda esta pergunta se na questão 11 a sua resposta foi "Sim") (R.M.)

( ) Automação de testes funcionais
( ) Teste de carga e stress
( ) Profiling (Perfil de execução)
( ) Cobertura
( ) Análise estática de código
( ) Obtenção de métricas de qualidade
( ) Análise do perfil de memória consumida / identificação de vazamentos de memória
( ) Outras. Quais? _____

13-13) Pretende utilizar ferramentas de testes? (Somente responda esta pergunta se na questão 11 a sua resposta foi "Não") (R.U.)

( ) Sim
( ) Não

14-14) Em quanto tempo pretende adotar uma ferramenta de testes? (Somente responda esta pergunta se na questão 13 a sua resposta foi "Sim") (R.U.)

( ) Em até 4 meses
( ) De 4 a 10 meses
( ) Em mais de 10 meses

15-15) Qual tipo de ferramenta pretende usar? (Somente responda esta pergunta se na questão 13 a sua resposta foi "Sim") (R.M.)

( ) Automação de testes funcionais
( ) Teste de carga e stress
( ) Profiling (Perfil de execução)
( ) Cobertura
( ) Análise estática de código
( ) Obtenção de métricas de qualidade
( ) Análise do perfil de memória consumida / identificação de vazamentos de memória
( ) Outras. Quais? _____

16-16) Porque não utiliza ferramentas de testes? (Somente responda esta pergunta se na questão 11 a sua resposta foi "Não") (R.M.)

( ) Preço elevado
( ) Tecnologia precária
( ) Agrega pouco valor
( ) Não auxilia muito os testes
( ) Aumenta o esforço necessário para o teste
( ) Os resultados não justificam o investimento
( ) Não tenho necessidade
( ) Outros. Quais? _____

17-17) Qual o tamanho da Área de TI na empresa em que trabalha? (R.U.)

( ) Até 10 pessoas
( ) De 10 a 30 pessoas
( ) De 30 a 100 pessoas
( ) Mais de 100 pessoas

18-18) Qual é o objetivo do desenvolvimento de software realizado na empresa em que trabalha? (R.U.)

( ) Suportar os negócios da própria empresa
( ) Venda, como um produto final
( ) Adaptação de pacotes/soluções em software e implantação em clientes
( ) Outro. Qual? _____

19-19) Quem compra o software desenvolvido na empresa em que trabalha? (Somente responda esta pergunta se na questão 18 a sua resposta foi a alternativa "Venda") (R.U.)

( ) Pessoa Física

( ) Pessoa Jurídica
( ) Ambos

20-20) Qual o percentual do tempo total de desenvolvimento de um sistema que é investido, na média, em atividades de verificação e testes nos desenvolvimentos de software?

_____

21-21) Qual é, na sua opinião e na média, o nível de qualidade das aplicações desenvolvidas? (Nota de 1 a 10, onde 1 é a menor nota) (R.U.)

( ) 1
( ) 2
( ) 3
( ) 4
( ) 5
( ) 6
( ) 7
( ) 8
( ) 9
( ) 10

22-22) Qual é, na sua opinião e na média, a percepção de qualidade dos usuários com relação às aplicações desenvolvidas? (Nota de 1 a 10, onde 1 é a menor nota) (R.U.)

( ) 1
( ) 2
( ) 3
( ) 4
( ) 5
( ) 6
( ) 7
( ) 8
( ) 9
( ) 10

23-23) Qual é, na sua opinião e na média, o nível de satisfação dos usuários das aplicações desenvolvidas? (R.U.)

( ) Muito Alto
( ) Alto
( ) Médio
( ) Baixo
( ) Muito Baixo

Parte 2 – Avaliação do qualidade dos processos de teste

Categoria Critério - Planejamento do Teste

24-1) Os testadores identificam os riscos técnicos e de negócio associados à implementação e operação do software? (R.U.)

( ) Sim
( ) Não

25-2) Estes riscos são endereçados no plano de testes? (R.U.)

( ) Sim
( ) Não

26-3) É criado um plano de testes para cada sistema desenvolvido? (R.U.)

( ) Sim
( ) Não

27-4) O plano identifica os requisitos de teste e objetivos? (R.U.)

( ) Sim
( ) Não

28-5) O plano estabelece critérios de sucesso para cada requisito? (R.U.)

( ) Sim
( ) Não

29-6) Os usuários são envolvidos ativamente no desenvolvimento do plano de testes? (R.U.)

( ) Sim
( ) Não

30-7) Os plano de testes inclui o envolvimento do usuário no teste? (R.U.)

( ) Sim
( ) Não

31-8) Os planos de teste são cumpridos? (R.U.)

( ) Sim
( ) Não

32-9) É necessária aprovação para desvios em relação ao plano de testes? (R.U.)

( ) Sim
( ) Não

33-10) Os planos de testes são alterados para refletir mudanças em abordagens de testes e mudanças nos requisitos de software? (R.U.)

( ) Sim
( ) Não

34-11) O plano de teste estabelece o(s) critério(s) que o(s) software(s) precisa(m) satisfazer para ser colocado em produção? (R.U.)

( ) Sim
( ) Não

35-12) O usuário concorda com este(s) critério(s)? (R.U.)

( ) Sim
( ) Não

Categoria Critério - Suporte gerencial para o teste

36-1) A gerência provê os recursos necessários (incluindo tempo) para treinar, planejar, conduzir, e avaliar os resultados relacionados às atribuições de teste de software adequadamente? (R.U.)

( ) Sim
( ) Não

37-2) Os testadores são envolvidos, desde o início até o término do projeto de desenvolvimento de software, para garantir que as considerações relevantes aos testes são continuamente endereçadas? (R.U.)

( ) Sim
( ) Não

38-3) A gerência aloca tantos recursos quanto necessário para testar os processos e ferramentas relacionados aos testes, assim como, aloca para ferramentas e processos relacionados ao desenvolvimento? (R.U.)

( ) Sim
( ) Não

39-4) A gerência gasta tanto tempo pessoal para planejar o teste e para executá-lo quanto gasta com o planejamento do desenvolvimento do software e sua execução? (R.U.)

( ) Sim
( ) Não

40-5) A gerência é capacitada e suficientemente treinada em teoria de teste, processos e ferramentas para gerenciar o planejamento e a execução dos testes eficientemente, bem como entender e agir em cima dos resultados dos testes eficientemente? (R.U.)

( ) Sim
( ) Não

Categoria Critério - Processos de Teste

41-1) Os testadores seguem os processos de planejar os testes, preparar a massa de testes, executar os testes, desenvolver os testes e reportar seus resultados? (R.U.)

( ) Sim
( ) Não

42-2) Os processos documentados de teste podem ser corretamente interpretados pelos testadores de forma que os procedimentos de testes podem ser seguidos como pretendido durante sua utilização? (R.U.)

( ) Sim
( ) Não

43-3) Os processos utilizados para testar cobrem todas as atividades que são necessárias para exercer um teste efetivo? (R.U.)

( ) Sim
( ) Não

44-4) Algum plano foi desenvolvido e colocado em prática para amadurecer os processos de teste de forma que eles se tornem mais eficientes, efetivos, e pontuais (dentro do tempo esperado e/ou necessário)? (R.U.)

( ) Sim
( ) Não

45-5) Os usuários dos processos de testes (testadores) constroem os processos utilizados para testar? (R.U.)

( ) Sim
( ) Não

Categoria Critério - Ferramentas de Teste

46-1) Os testadores utilizam uma ferramenta automatizada para gerar e reutilizar dados de teste? (R.U.)

( ) Sim
( ) Não

47-2) As ferramentas de teste são escolhidas de uma maneira lógica? (Ou seja, as necessidades de teste guiam a procura e a aquisição das ferramentas) (R.U.)

( ) Sim
( ) Não

48-3) Os testadores podem somente utilizar ferramentas de testes após terem recebidos treinamento adequado sobre como utilizar ferramentas de testes? (R.U.)

( ) Sim
( ) Não

49-4) A utilização da ferramenta de teste é especificada no plano de teste? (Ou seja, a utilização das ferramentas é mandatória, não opcional, ou é de acordo com a vontade do testador?) (R.U.)

( ) Sim
( ) Não

50-5) O processo de suporte na utilização de ferramentas de teste foi estabelecido? (R.U.)

( ) Sim
( ) Não

51-6) Este processo provê aos testadores as informações e instruções necessárias? (R.U.)

( ) Sim
( ) Não

Categoria Critério - Treinamento em Teste

52-1) Existe uma plano de treinamento para os testadores e este é utilizado para desenvolver um testador a partir de um estado de "ausência de capacitação" até um estado de "capacitação plena" ? (R.U.)

( ) Sim
( ) Não

53-2) Os testadores são treinados adequadamente em processos de teste antes de utilizá-los para testar? (R.U.)

( ) Sim
( ) Não

54-3) Os testadores são treinados em teoria de teste, análise de risco, as diferentes abordagens para testar, etc, de maneira que eles entendam "por quê" eles devem executar certas etapas de teste? (R.U.)

( ) Sim
( ) Não

55-4) Os testadores são treinados em estatísticas, de forma a entender o nível de confiança que eles podem prover ao usuário através de diferentes abordagens de testes e como interpretar os resultados dos testes? (R.U.)

( ) Sim
( ) Não

56-5) Os testadores são treinados em como mensurar a performance do processo, e como eles devem utilizar os resultados destas medidas para melhorar o processo de teste? (R.U.)

( ) Sim
( ) Não

<u>Categoria Critério - Satisfação do Usuário</u>

57-1) Os usuários obtém as informações que necessitam para acompanhar o progresso dos testes e avaliar os resultados antes de colocar o software em produção? (R.U.)

( ) Sim
( ) Não

58-2) Questionários são aplicados aos usuários para avaliar a sua satisfação com o planejamento do teste, a execução do teste, resultados dos testes, comunicações, etc? (R.U.)

( ) Sim
( ) Não

59-3) Os usuários participam nos testes que determinam se o software está aceitável para utilização ou não? (R.U.)

( ) Sim
( ) Não

60-4) O plano para testar a aplicação é apresentado aos usuários para aprovação, e, a aprovação deles considera os testes satisfatórios no caso do plano ser executado? (R.U.)

( ) Sim
( ) Não

61-5) Os usuários validam atividades tais como entrada de dados, utilização da saída, utilização de terminal, utilização de manual, etc, como parte do teste? (R.U.)

( ) Sim

( ) Não

Categoria Critério - Eficiência do Teste

62-1) O teste é planejado de forma que os recursos de teste sejam alocados para garantir que os maiores riscos sejam endereçados antes dos menores? (R.U.)

( ) Sim
( ) Não

63-2) Existe um processo de mensurar a eficiência dos processos de teste? (R.U.)

( ) Sim
( ) Não

64-3) A compatibilidade com o orçamento e a agenda são examinadas e as diferenças são endereçadas efetivamente? (R.U.)

( ) Sim
( ) Não

65-4) A utilização de ferramenta é mensurada para avaliar a contribuição trazida pelo teste automatizado? (R.U.)

( ) Sim
( ) Não

66-5) O percentual de defeitos removido versus o total de defeitos eventualmente atribuídos a fase de desenvolvimento é mensurado? (R.U.)

( ) Sim
( ) Não

Categoria Critério - Controle de Qualidade do Teste

67-1) Os defeitos encontrados pelos testadores durante o teste são documentados e endereçados efetivamente? (R.U.)

( ) Sim
( ) Não

68-2) O plano de testes é revisado e inspecionado durante e depois de sua elaboração por pares (peer review) para adequação e compatibilização aos padrões/normas de testes? (R.U.)

( ) Sim
( ) Não

69-3) O plano de testes inclui procedimentos que serão utilizados para verificar se o plano é executado de acordo com o planejado? (R.U.)

( ) Sim
( ) Não

70-4) Relatórios são preparados regularmente para mostra o status completo do teste de um sistema específico? (R.U.)

( ) Sim

( ) Não

71-5) Os relatórios de controle de qualidade específicos (de sistemas pontuais) são sumarizados periodicamente para mostrar a eficiência e a efetividade do teste em todos os sistemas de informação da organização? (R.U.)

( ) Sim
( ) Não

Parte 3 - Abrangência dos testes

72-1) Suportabilidade (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

73-2) Eficiência (Efficiency) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

74-3) Acurácia (Accuracy) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

75-4) Performance (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

76-5) Robustez (Robustness) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes

( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

77-6) Conformidade (Correctness) (R.U.)
( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

78-7) Confiabilidade (Reliability) (R.U.)
( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

79-8) Integridade (Integrity) (R.U.)
( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

80-9) Usabilidade (Usability) (R.U.)
( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

81-10) Manutenabilidade (Maintainability) (R.U.)
( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

82-11) Testabilidade (Testability) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

83-12) Flexibilidade (Flexibility) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

84-13) Portabilidade (Portability) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

85-14) Reusabilidade (Reusability) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

86-15) Interoperabilidade (Interoperability) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar
( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

87-16) Segurança (Security) (R.U.)

( ) Não conhece
( ) Conhece mas não utiliza
( ) Conhece e utiliza informalmente algumas vezes
( ) Conhece e utiliza informalmente sempre que se aplicar

( ) Conhece e utiliza formalmente algumas vezes
( ) Conhece e utiliza formalmente sempre que se aplicar
( ) Não se aplica. Por que? _____

APÊNDICE 2 – GRÁFICOS UTILIZADO NA ANÁLISE DOS DADOS COLETADOS

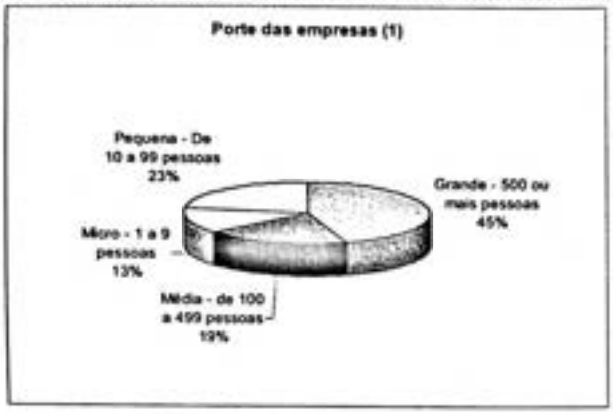

Gráfico 1 - Porte das empresas

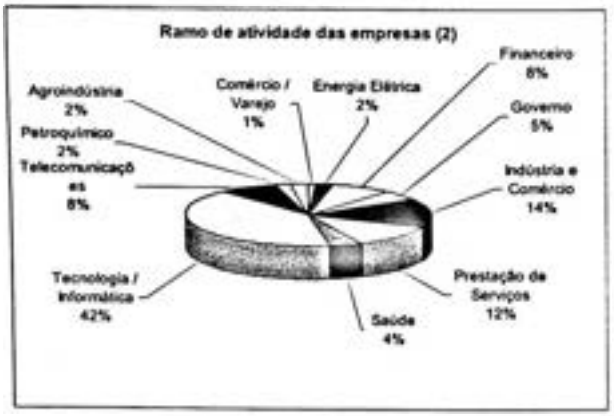

Gráfico 2 - Ramo de atividade das empresas

Gráfico 3 - Realização do desenvolvimento de sistemas

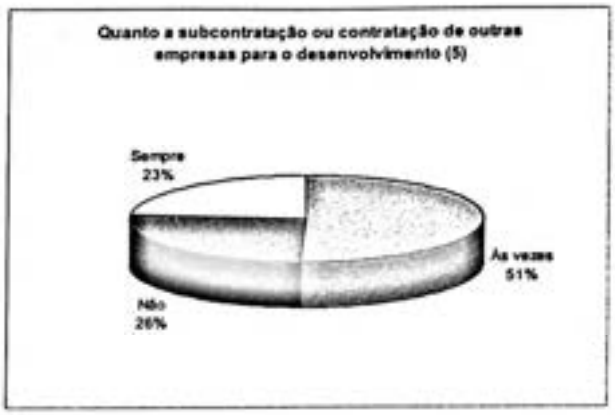

Gráfico 4 - Quanto a subcontratação ou contratação de outras empresas para o desenvolvimento

Gráfico 5 - Quanto ao tipo de processo de desenvolvimento de software

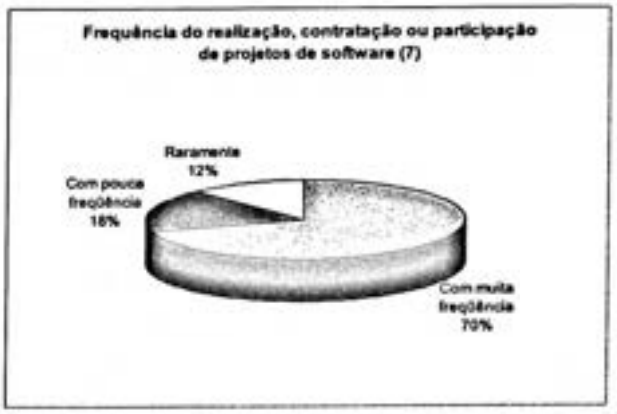

Gráfico 6 - Frequência do realização, contratação ou participação de projetos de software

Gráfico 7 - Existência de algum tipo de validação ou teste para avaliar a qualidade do software, seja ele comprado ou desenvolvido internamente

Gráfico 8 - Quem realiza dos testes?

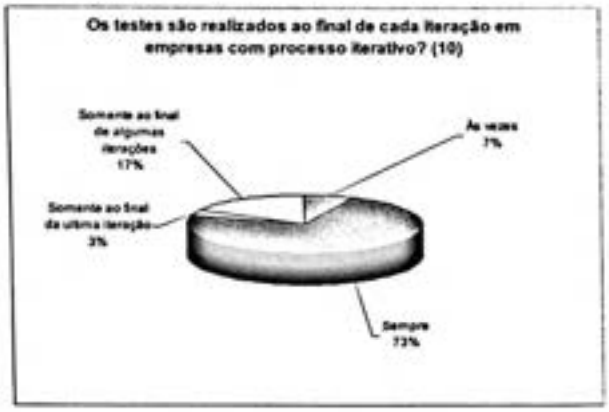

Gráfico 9 - Os testes são realizados ao final de cada iteração em empresas com processo iterativo?

Gráfico 10 - Os testes são realizados com auxílio de alguma ferramenta?

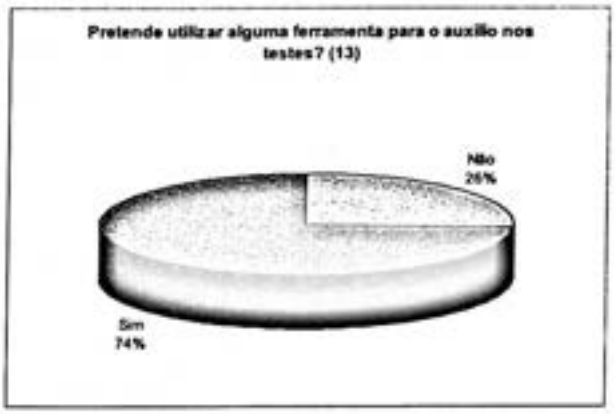

Gráfico 11 - Pretende utilizar alguma ferramenta para o auxílio nos testes?

Gráfico 12 - Em quanto tempo pretende adotar uma ferramenta?

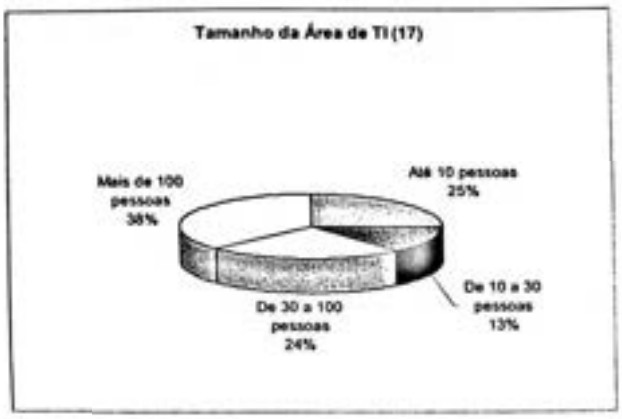

Gráfico 13 - Tamanho da Área de TI

Gráfico 14 - Objetivos do software desenvolvido

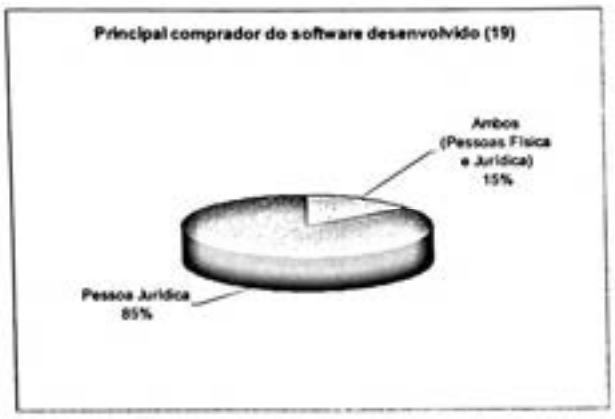

Gráfico 15 - Principal comprador do software desenvolvido

Gráfico 16 - Por que não utiliza uma ferramenta de testes?

Gráfico 17 - Qual tipo de ferramenta pretende usar?

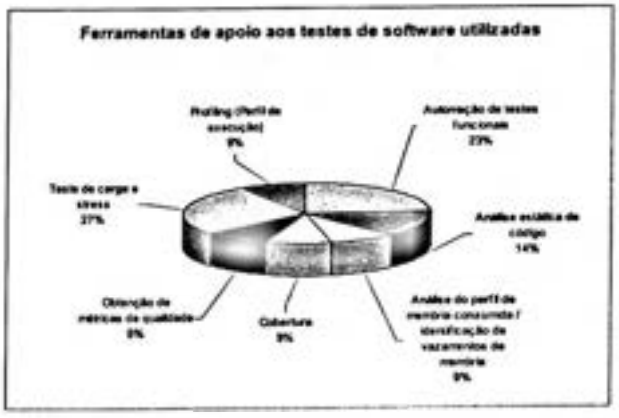

Gráfico 18 - Ferramentas de apoio aos testes de software utilizadas

Gráfico 19 - Porte das empresas que realizam desenvolvimento de Sistemas de Informação

Gráfico 20 - Ramo das empresas que realizam desenvolvimento de Sistemas de Informação

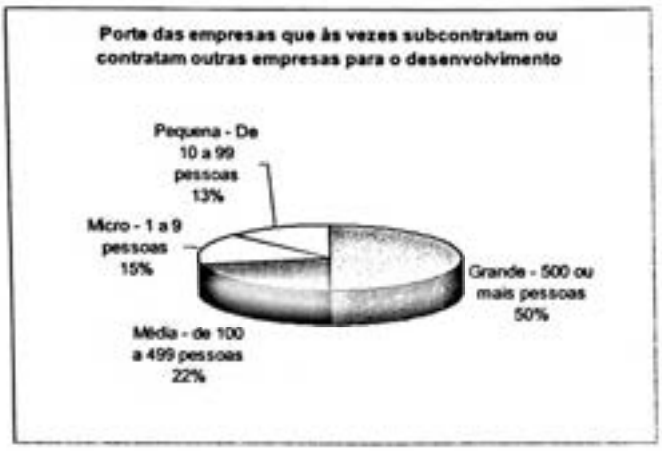

Gráfico 21 - Porte das empresas que às vezes subcontratam ou contratam outras empresas para o desenvolvimento

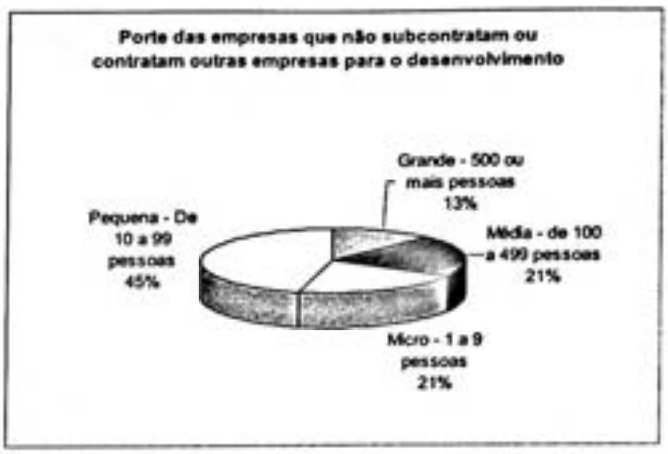

Gráfico 22 - Porte das empresas que não subcontratam ou contratam outras empresas para o desenvolvimento

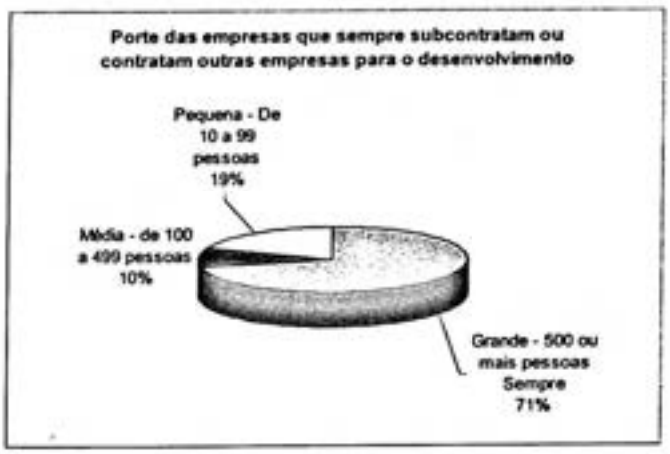

Gráfico 23 - Porte das empresas que sempre subcontratam ou contratam outras empresas para o desenvolvimento

Gráfico 24 - Processo de desenvolvimento nas empresas de grande porte

Gráfico 25 - Processo de desenvolvimento nas empresas de médio porte

Gráfico 26 - Processo de desenvolvimento nas empresas de pequeno porte

Gráfico 27 - Processo de desenvolvimento nas micro empresas

Gráfico 28 - Processo de desenvolvimento na Agroindústria

Gráfico 29 - Processo de desenvolvimento no ramo de Energia Elétrica

Gráfico 30 - Processo de desenvolvimento no ramo Financeiro

Gráfico 31 - Processo de desenvolvimento no setor governamental

Gráfico 32 - Processo de desenvolvimento na Indústria e Comércio

Gráfico 33 - Processo de desenvolvimento no setor de Prestação de Serviços

Gráfico 34 - Processo de desenvolvimento no setor de Saúde

Gráfico 35 - Processo de desenvolvimento no setor de Tecnologia e Informática

Gráfico 36 - Processo de desenvolvimento no setor de Telecomunicações

Gráfico 37 - Processo de desenvolvimento em empresas com até 10 pessoas na área de TI

Gráfico 38 - Processo de desenvolvimento em empresas com 10 a 30 pessoas na área de TI

Gráfico 39 - Processo de desenvolvimento em empresas com 30 a 100 pessoas na área de TI

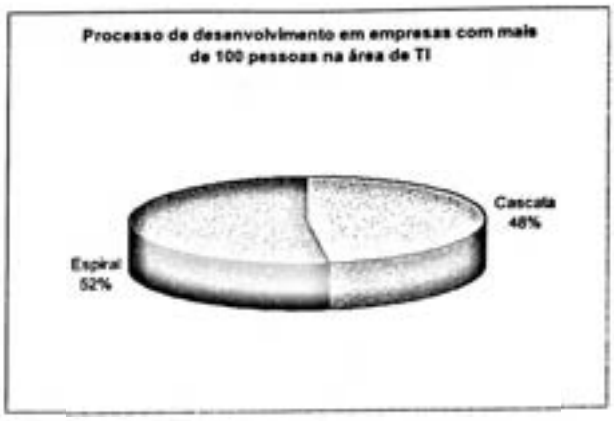

Gráfico 40 - Processo de desenvolvimento em empresas com mais de 100 pessoas na área de TI

Gráfico 41 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em empresas de grande porte

Gráfico 42 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em empresas de médio porte

Gráfico 43 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvidoem empresas de pequeno porte

Gráfico 44 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em micro empresas

Gráfico 45 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido na Agroindústria

Gráfico 46 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido no setor de Energia Elétrica

Gráfico 47 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido no setor Financeiro

Gráfico 48 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido no setor Governamental

Gráfico 49 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido na Indústria e Comercio

Gráfico 50 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido no ramo de Prestação de Serviços

Gráfico 51 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido no setor de Saúde

Gráfico 52 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido no setor de Tecnologia e Informática

Gráfico 53 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido no setor de Telecomunicações

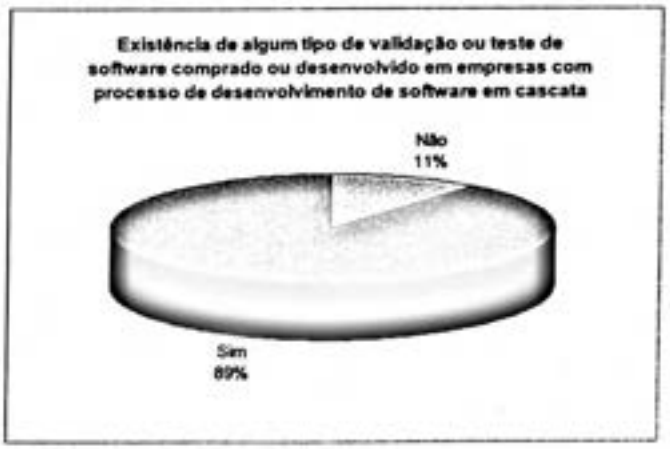

Gráfico 54 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em empresas com processo de desenvolvimento de software em cascata

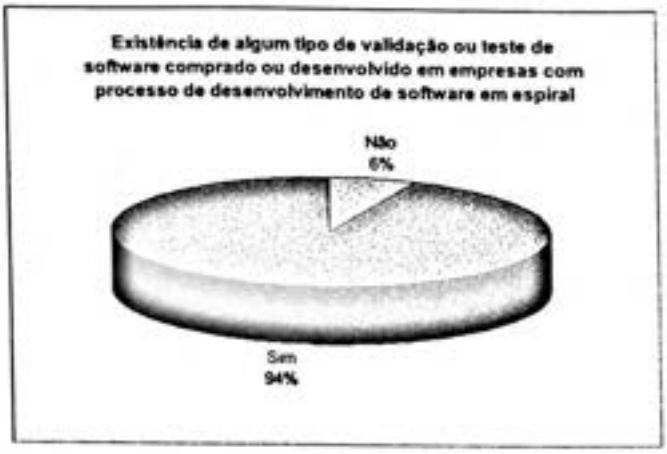

Gráfico 55 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em empresas com processo de desenvolvimento de software em espiral

Gráfico 56 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em empresas com até 10 pessoas na área de TI

Gráfico 57 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em empresas com 10 a 30 pessoas na área de TI

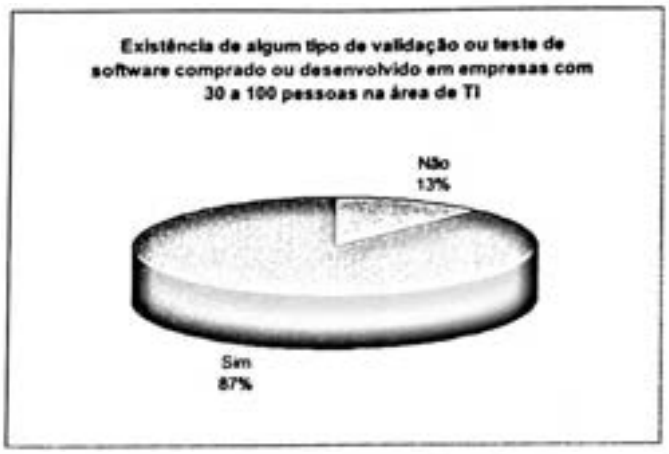

Gráfico 58 - Existência de algum tipo de validação ou teste de software comprado ou desenvolvido em empresas com 30 a 100 pessoas na área de TI

Gráfico 59 - Utilização de ferramentas de auxílio ao teste de software em empresas de grande porte

Gráfico 60 - Utilização de ferramentas de auxílio ao teste de software em empresas de médio porte

Gráfico 61 - Utilização de ferramentas de auxílio ao teste de software em empresas de pequeno porte

Gráfico 62 - Utilização de ferramentas de auxílio ao teste de software em micro empresas

Gráfico 63 - Utilização de ferramentas de auxílio ao teste de software na Agroindústria

Gráfico 64 - Utilização de ferramentas de auxílio ao teste de software no setor de Energia Elétrica

Gráfico 65 - Utilização de ferramentas de auxílio ao teste de software no setor Financeiro

Gráfico 66 - Utilização de ferramentas de auxílio ao teste de software no setor Governamental

Gráfico 67 - Utilização de ferramentas de auxílio ao teste de software na Indústria e Comércio

Gráfico 68 - Utilização de ferramentas de auxílio ao teste de software no setor de Prestação de Serviços

Gráfico 69 - Utilização de ferramentas de auxílio ao teste de software no setor de Saúde

Gráfico 70 - Utilização de ferramentas de auxílio ao teste de software no setor de Tecnologia e Informática

Gráfico 71 - Utilização de ferramentas de auxílio ao teste de software no setor de Telecomunicações

Gráfico 72 - Utilização de ferramentas de auxílio ao teste de software em empresas com o processo de desenvolvimento de software em cascata

Gráfico 73 - Utilização de ferramentas de auxílio ao teste de software em empresas com o processo de desenvolvimento de software em espiral

Gráfico 74 - Utilização de ferramentas de auxílio ao teste de software em empresas com até 10 pessoas na área de TI

Gráfico 75 - Utilização de ferramentas de auxílio ao teste de software em empresas de 10 a 30 pessoas na área de TI

Gráfico 76 - Utilização de ferramentas de auxílio ao teste de software em empresas de 30 a 100 pessoas na área de TI

Gráfico 77 - Utilização de ferramentas de auxílio ao teste de software em empresas com mais de 100 pessoas na área de TI

Gráfico 78 - Intenção em utilizar ferramenta de apoio aos testes na Agroindústria

Gráfico 79 - Intenção em utilizar ferramenta de apoio aos testes no setor de Energia Elétrica

Gráfico 80 - Intenção em utilizar ferramenta de apoio aos testes no setor Financeiro

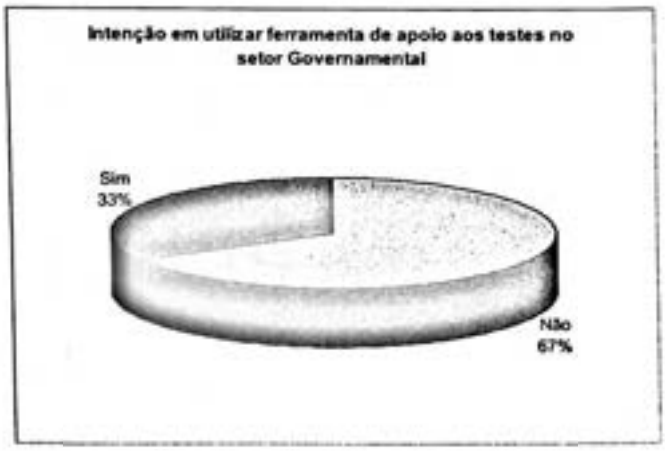

Gráfico 81 - Intenção em utilizar ferramenta de apoio aos testes no setor Governamental

Gráfico 82 - Intenção em utilizar ferramenta de apoio aos testes na Indústria e Comércio

Gráfico 83 - Intenção em utilizar ferramenta de apoio aos testes no setor de Prestação de Serviços

Gráfico 84 - Intenção em utilizar ferramenta de apoio aos testes no setor de Saúde

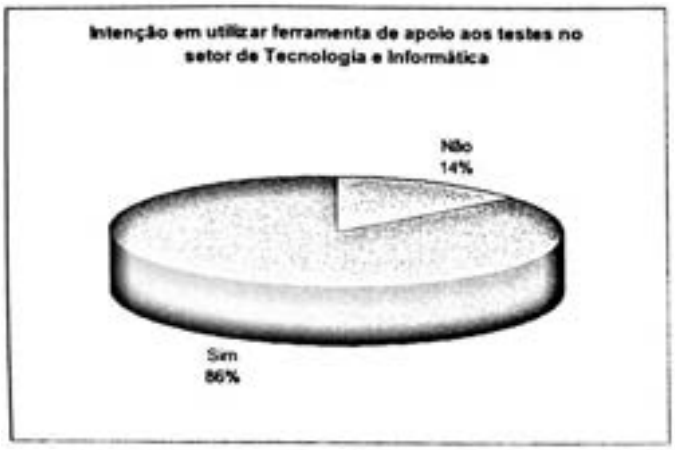

Gráfico 85 - Intenção em utilizar ferramenta de apoio aos testes no setor de Tecnologia e Informática

Gráfico 86 - Intenção em utilizar ferramenta de apoio aos testes no setor de Telecomunicações

Gráfico 87 - Tamanho da Área de TI na Agroindústria

Gráfico 88 - Tamanho da Área de TI no setor de Energia Elétrica

Gráfico 89 - Tamanho da Área de TI no setor Financeiro

Gráfico 90 - Tamanho da Área de TI no setor Governamental

Gráfico 91 - Tamanho da Área de TI na Indústria e Comércio

Gráfico 92 - Tamanho da Área de TI no setor de Prestação de Serviços

Gráfico 93 - Tamanho da Área de TI no setor de Prestação de Saúde

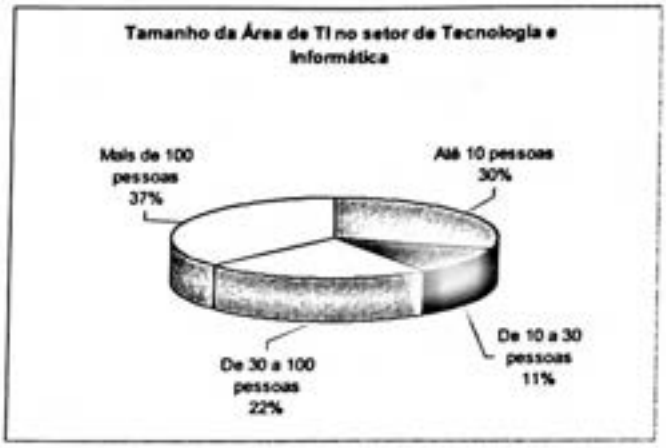

Gráfico 94 - Tamanho da Área de TI no setor de Tecnologia e Informática

Gráfico 95 - Tamanho da Área de TI no setor de Telecomunicações

Gráfico 96 - Realização de Testes ou Verificação em software comprado ou desenvolvido em empresas com foco em adaptação de pacotes/soluções em software e implantação em clientes

Gráfico 97 - Realização de Testes ou Verificação em software comprado ou desenvolvido em empresas com área de TI focada em suportar os próprios negócios da empresa

Gráfico 98 - Realização de Testes ou Verificação em software comprado ou desenvolvido em empresas com área de TI focada no desenvolvimento de software para venda como produto

Gráfico 99 - Utilização de ferramentas de apoio aos testes em empresas com foco em adaptação de pacotes/soluções em software e implantação em clientes

Gráfico 100 - Utilização de ferramentas de apoio aos testes em empresas com área de TI focada em suportar os próprios negócios da empresa

Gráfico 101 - Utilização de ferramentas de apoio aos testes em empresas com área de TI focada no desenvolvimento de software para venda como produto

Gráfico 102 - Ferramentas de apoio aos testes de software utilizadas em empresas de grande porte

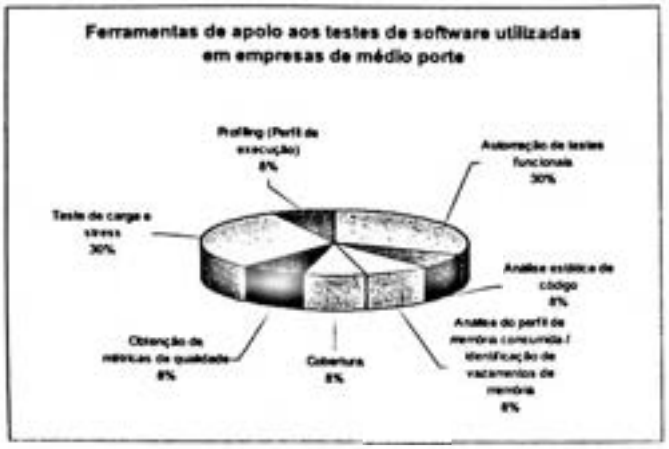

Gráfico 103 - Ferramentas de apoio aos testes de software utilizadas em empresas de médio porte

Gráfico 104 - Ferramentas de apoio aos testes de software utilizadas em empresas de pequeno porte

Gráfico 105 - Ferramentas de apoio aos testes de software utilizadas em empresas de pequeno porte

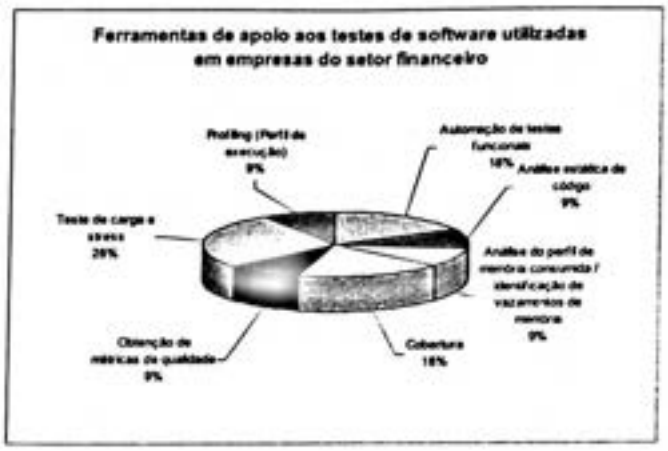

Gráfico 106 - Ferramentas de apoio aos testes de software utilizadas em empresas do setor financeiro

Gráfico 107 - Ferramentas de apoio aos testes de software utilizadas em empresas do setor governamental

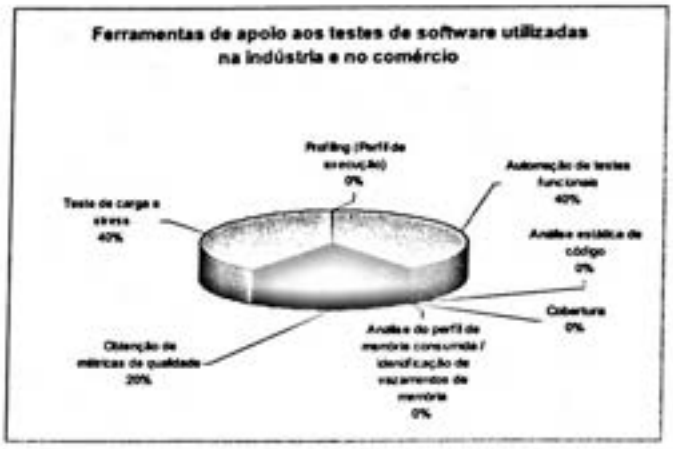

Gráfico 108 - Ferramentas de apoio aos testes de software utilizadas na indústria e no comércio

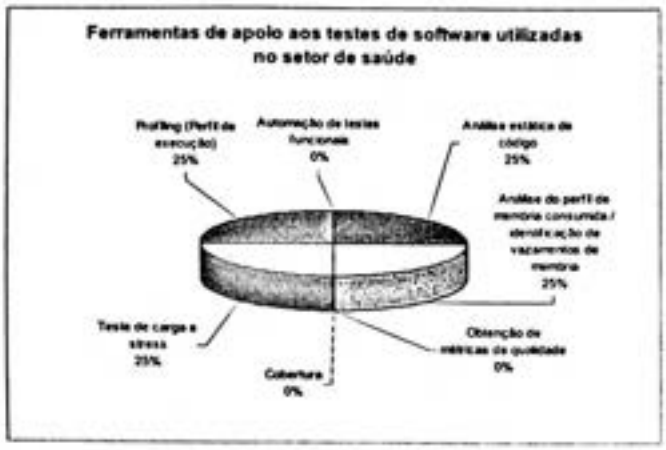

Gráfico 109 - Ferramentas de apoio aos testes de software utilizadas no setor de saúde

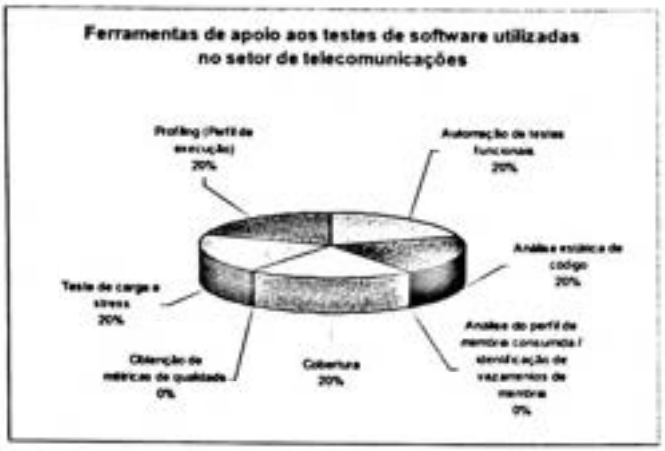

Gráfico 110 - Ferramentas de apoio aos testes de software utilizadas no setor de telecomunicações

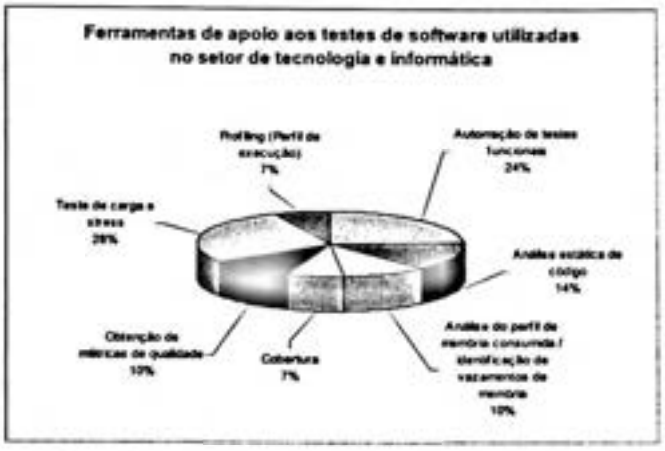

Gráfico 111 - Ferramentas de apoio aos testes de software utilizadas no setor de tecnologia e informática

Gráfico 112 - Ferramentas de apoio aos testes de software utilizadas nas empresas que utilizam um processo de desenvolvimento de software em cascata

Gráfico 113 - Ferramentas de apoio aos testes de software utilizadas nas empresas que utilizam um processo de desenvolvimento de software em espiral

Gráfico 114 - Ferramentas de apoio aos testes de software utilizadas em empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 115 - Ferramentas de apoio aos testes de software utilizadas em empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 116 - Ferramentas de apoio aos testes de software utilizadas em empresas onde a área de TI desenvolve software para venda como um produto final

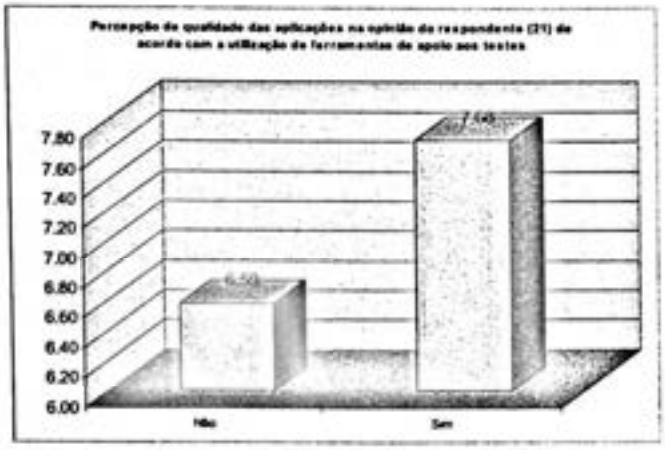

Gráfico 117 - Percepção de qualidade das aplicações na opinião do respondente de acordo com a utilização de ferramentas de apoio aos testes

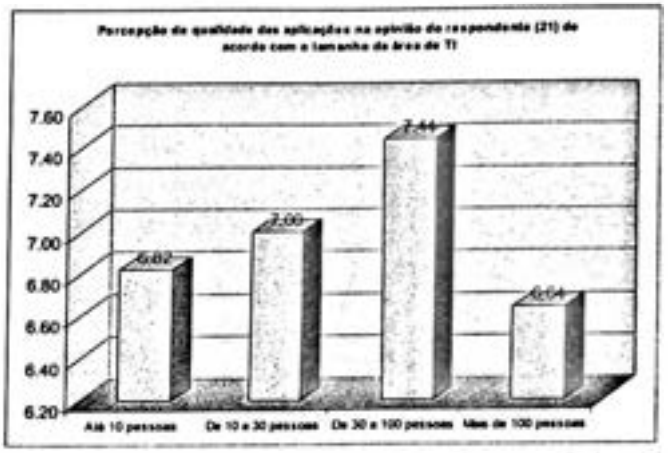

Gráfico 118 - Percepção de qualidade das aplicações na opinião do respondente de acordo com o tamanho da área de TI

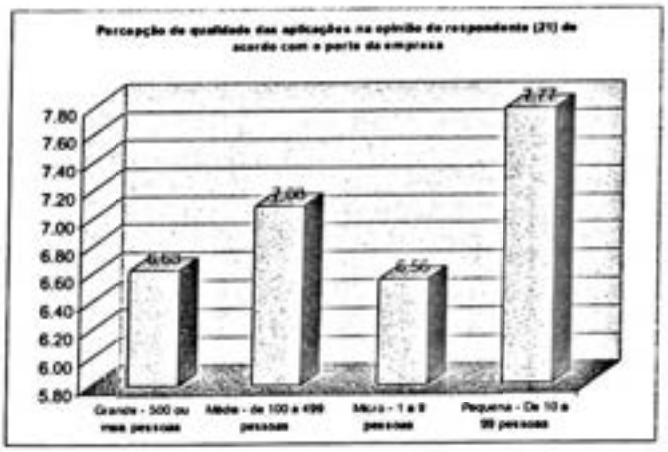

Gráfico 119 - Percepção de qualidade das aplicações na opinião do respondente de acordo com o porte da empresa

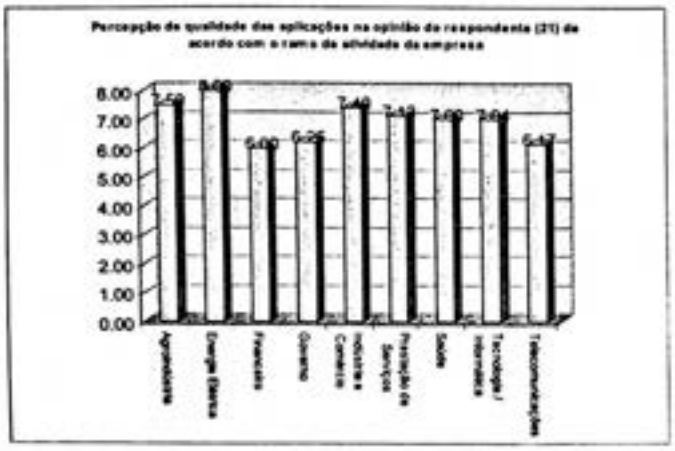

Gráfico 120 - Percepção de qualidade das aplicações na opinião do respondente de acordo com o ramo de atividade da empresa

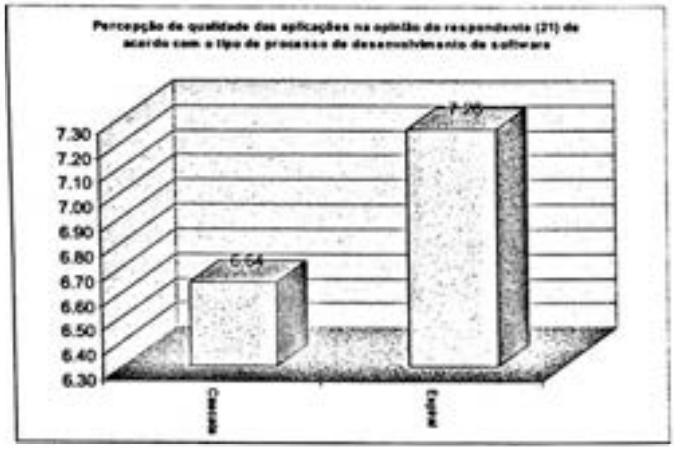

Gráfico 121 - Percepção de qualidade das aplicações na opinião do respondente de acordo com o tipo de processo de desenvolvimento de software

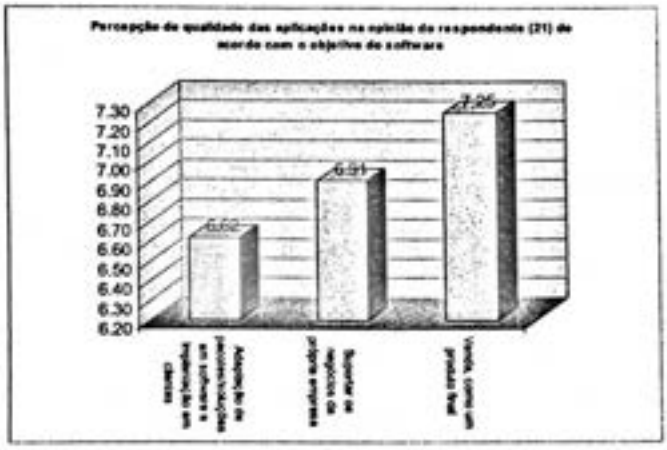

Gráfico 122 - Percepção de qualidade das aplicações na opinião do respondente de acordo com o objetivo do software

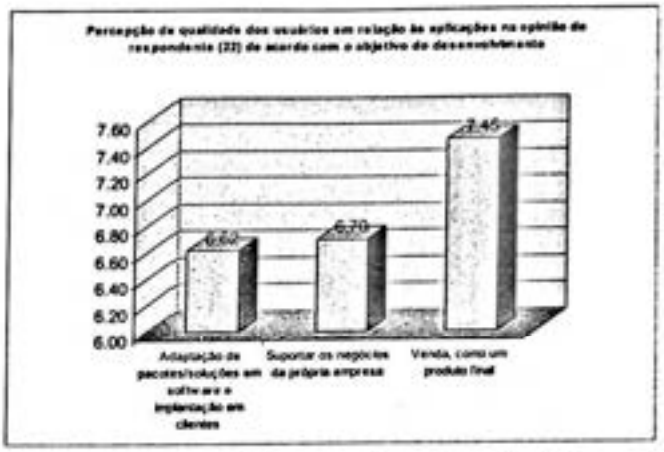

Gráfico 123 - Percepção de qualidade dos usuários em relação às aplicações na opinião do respondente de acordo com o objetivo do desenvolvimento

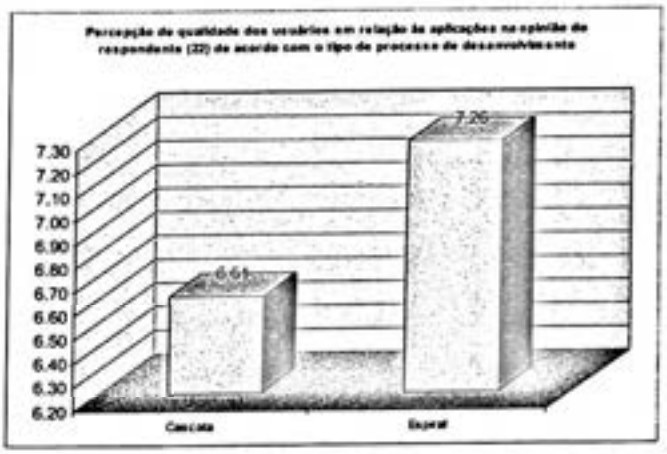

Gráfico 124 - Percepção de qualidade dos usuários em relação às aplicações na opinião do respondente de acordo com o tipo de processo de desenvolvimento

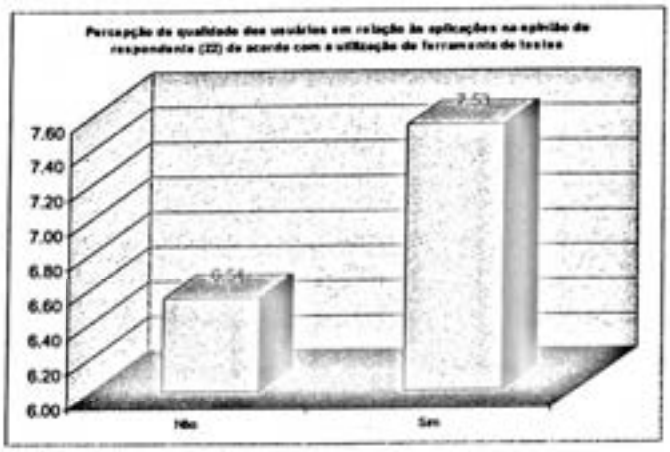

Gráfico 125 - Percepção de qualidade dos usuários em relação às aplicações na opinião do respondente de acordo com a utilização de ferramenta de testes

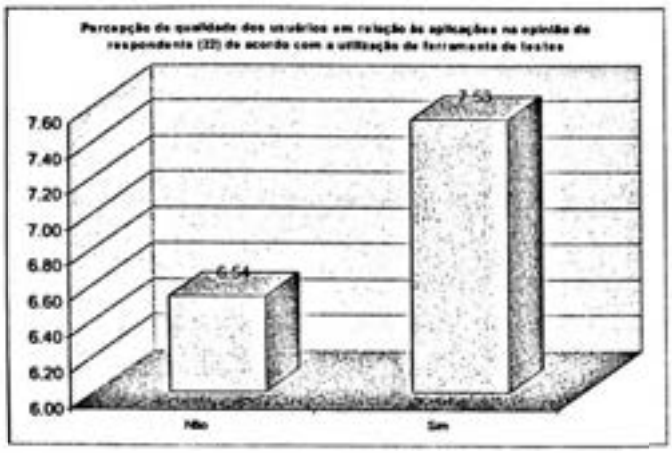

Gráfico 126 - Percepção de qualidade dos usuários em relação às aplicações na opinião do respondente de acordo com a utilização de ferramenta de testes

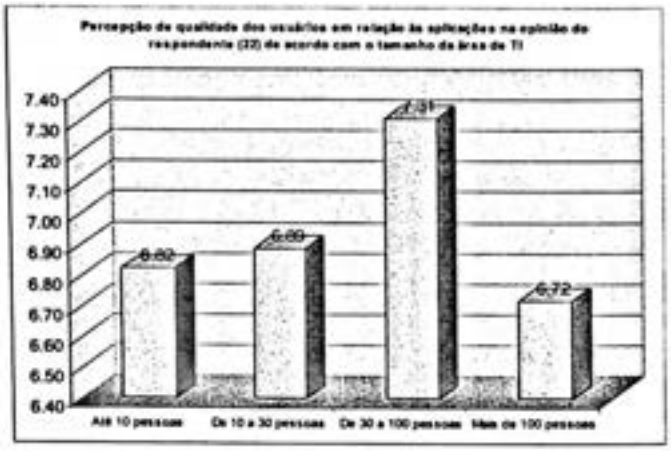

Gráfico 127 - Percepção de qualidade dos usuários em relação às aplicações na opinião do respondente de acordo com o tamanho da área de TI

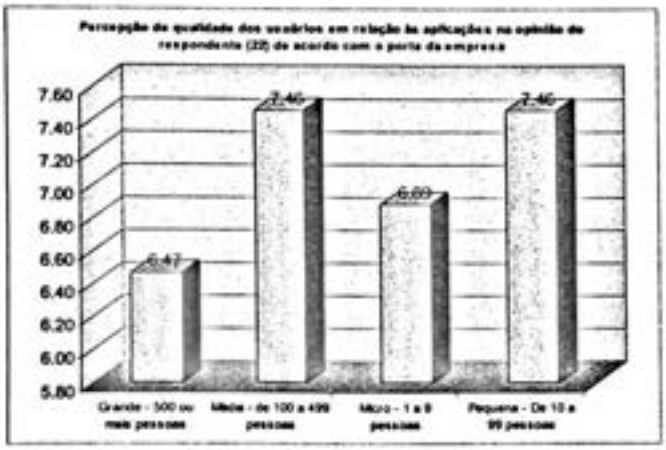

Gráfico 128 - Percepção de qualidade dos usuários em relação às aplicações na opinião do respondente de acordo com o porte da empresa

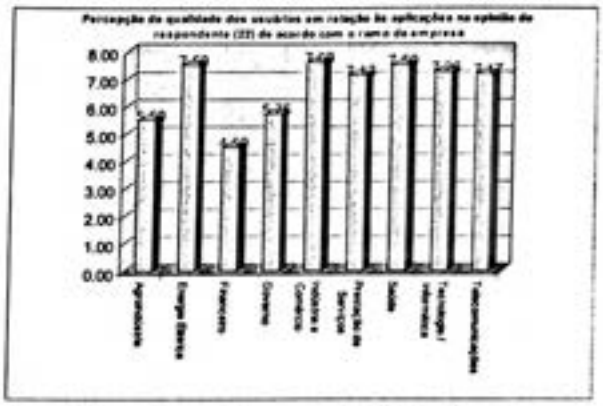

Gráfico 129 - Percepção de qualidade dos usuários em relação às aplicações na opinião do respondente de acordo com o ramo da empresa

Gráfico 130 - Nível de satisfação dos usuários na opinião dos respondentes

Gráfico 131 - Nível de satisfação dos usuários na opinião dos respondentes em empresas de grande porte

Gráfico 132 - Nível de satisfação dos usuários na opinião dos respondentes em empresas de médio porte

Gráfico 133 - Nível de satisfação dos usuários na opinião dos respondentes em micro empresas

Gráfico 134 - Nível de satisfação dos usuários na opinião dos respondentes em empresas de pequeno porte

Gráfico 135 - Nível de satisfação dos usuários na opinião dos respondentes na Agroindústria

Gráfico 136 - Nível de satisfação dos usuários na opinião dos respondentes no ramo de energia elétrica

Gráfico 137 - Nível de satisfação dos usuários na opinião dos respondentes no ramo financeiro

Gráfico 138 - Nível de satisfação dos usuários na opinião dos respondentes no governo

Gráfico 139 - Nível de satisfação dos usuários na opinião dos respondentes na indústria e no comércio

Gráfico 140 - Nível de satisfação dos usuários na opinião dos respondentes no ramo de prestação de serviços

Gráfico 141 - Nível de satisfação dos usuários na opinião dos respondentes no ramo de saúde

Gráfico 142 - Nível de satisfação dos usuários na opinião dos respondentes no ramo de tecnologia/informática

Gráfico 143 - Nível de satisfação dos usuários na opinião dos respondentes no ramo de telecomunicações

Gráfico 144 - Nível de satisfação dos usuários na opinião dos respondentes em área de TI de até 10 pessoas

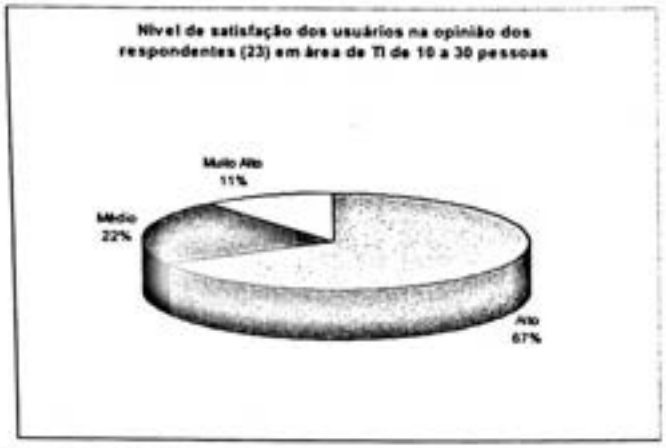

Gráfico 145 - Nível de satisfação dos usuários na opinião dos respondentes em área de TI de 10 a 30 pessoas

Gráfico 146 - Nível de satisfação dos usuários na opinião dos respondentes em área de TI de 30 a 100 pessoas

Gráfico 147 - Nível de satisfação dos usuários na opinião dos respondentes em área de TI com mais de 100 pessoas

Gráfico 148 - Nível de satisfação dos usuários na opinião dos respondentes em empresas que não utilizam ferramentas para teste de software

Gráfico 149 - Nível de satisfação dos usuários na opinião dos respondentes em empresas que não utilizam ferramentas para teste de software

Gráfico 150 - Nível de satisfação dos usuários na opinião dos respondentes em empresas com o processo de desenvolvimento de software em cascata

Gráfico 151 - Nível de satisfação dos usuários na opinião dos respondentes em empresas com o processo de desenvolvimento de software em espiral

Gráfico 152 - Nível de satisfação dos usuários na opinião dos respondentes em empresas que realizam adaptação de pacotes/soluções em software e implantação em clientes

Gráfico 153 - Nível de satisfação dos usuários na opinião dos respondentes em empresas que desenvolvem software para suportar os negócios da própria empresa

Gráfico 154 - Nível de satisfação dos usuários na opinião dos respondentes em empresas que desenvolvem software para vendê-lo como um produto

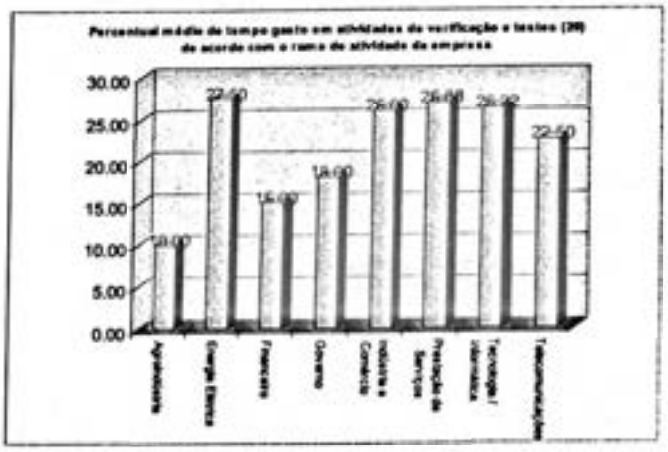

Gráfico 155 - Percentual médio de tempo gasto em atividades de verificação e testes de acordo com o ramo de atividade da empresa

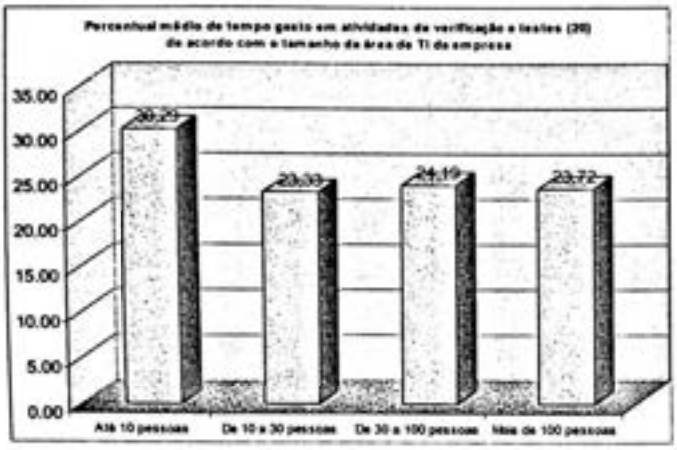

Gráfico 156 - Percentual médio de tempo gasto em atividades de verificação e testes de acordo com o tamanho da área de TI da empresa

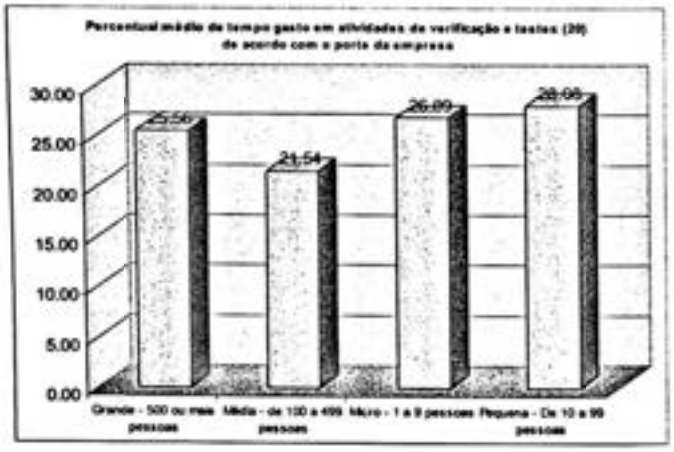

Gráfico 157 - Percentual médio de tempo gasto em atividades de verificação e testes de acordo com o porte da empresa

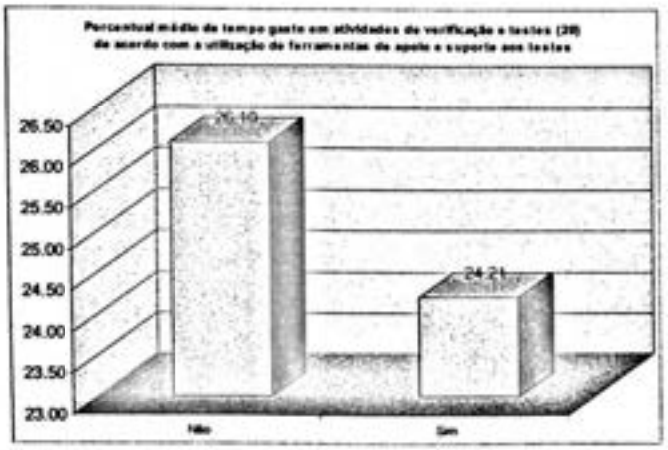

Gráfico 158 - Percentual médio de tempo gasto em atividades de verificação e testes de acordo com a utilização de ferramentas de apoio e suporte aos testes

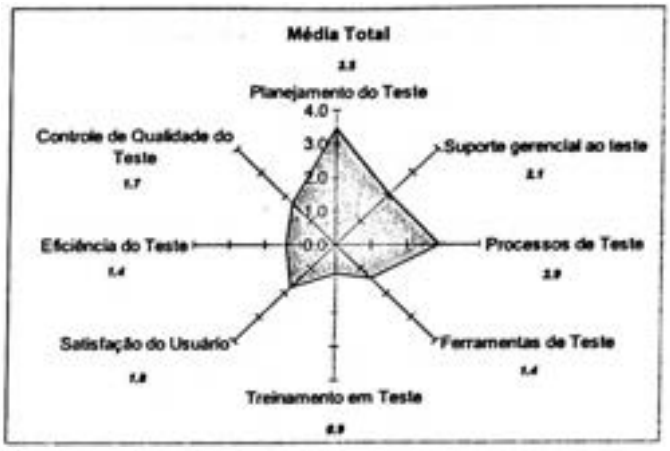

Gráfico 159 - Média Total

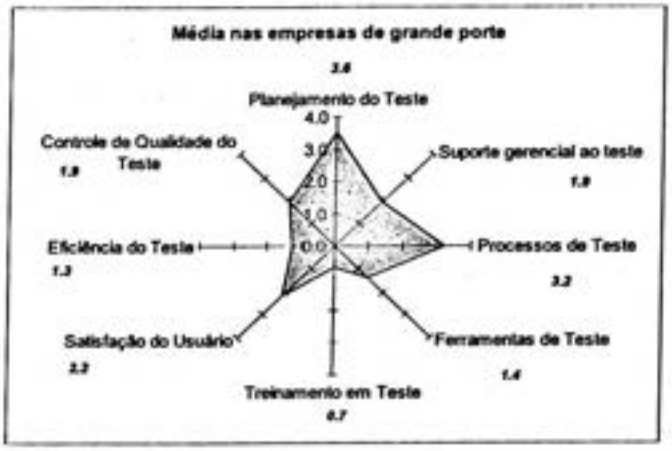

Gráfico 160 - Média nas empresas de grande porte

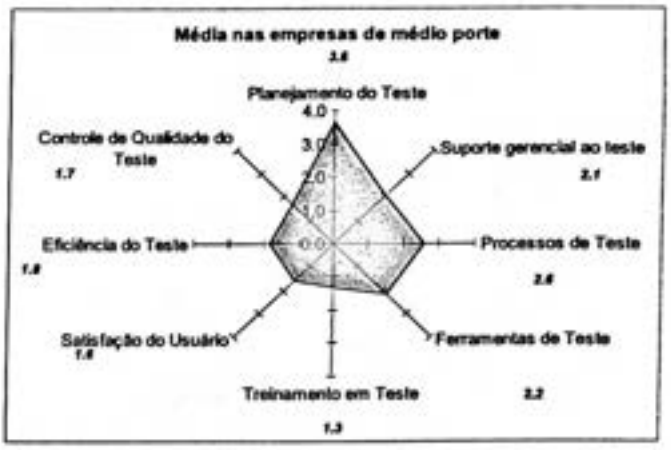

Gráfico 161 - Média nas empresas de médio porte

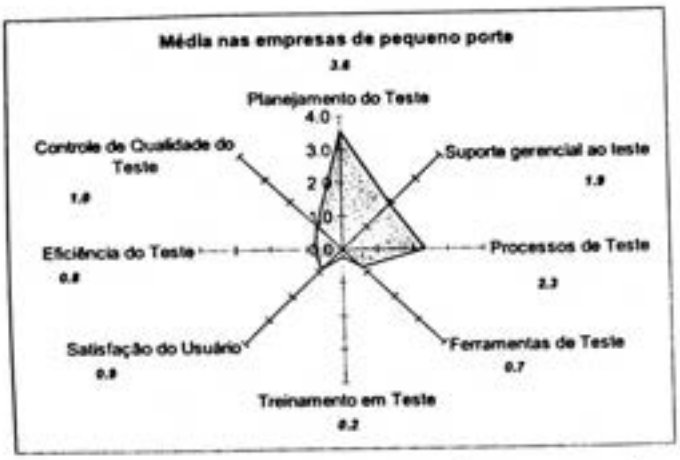

Gráfico 162 - Média nas empresas de pequeno porte

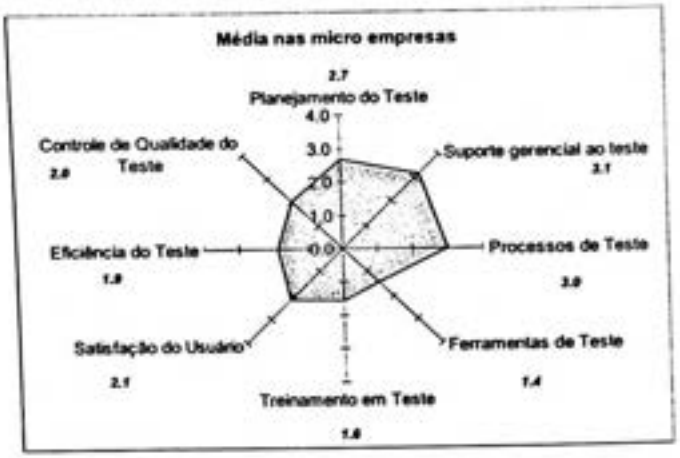

Gráfico 163 - Média nas micro empresas

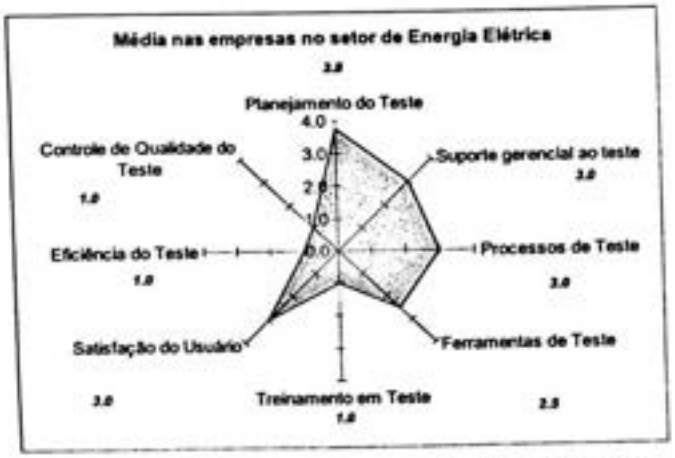

Gráfico 164 - Média nas empresas no setor de Energia Elétrica

Gráfico 165 - Média nas empresas no setor Financeiro

Gráfico 166 - Média nas empresas no setor Governamental

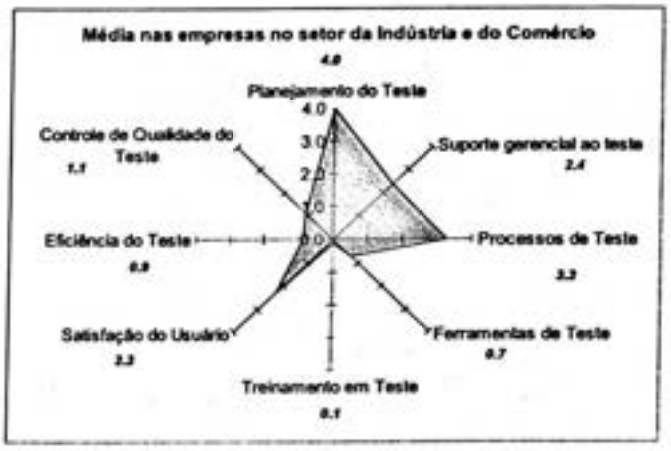

Gráfico 167 - Média nas empresas no setor da Indústria e do Comércio

Gráfico 168 - Média nas empresas no setor de Prestação de Serviços

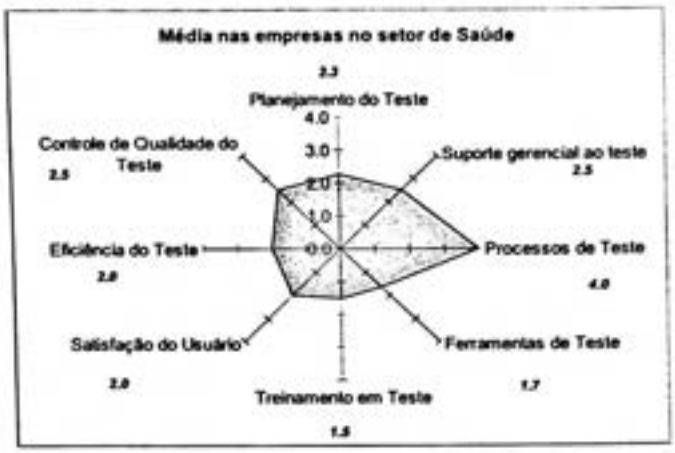

Gráfico 169 - Média nas empresas no setor de Saúde

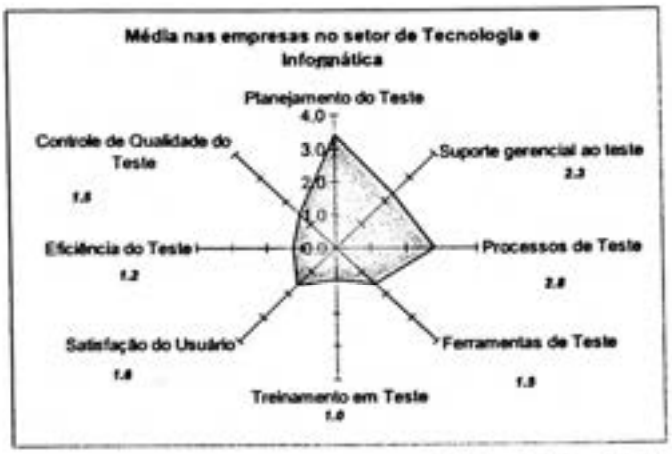

Gráfico 170 - Média nas empresas no setor de Tecnologia e Informática

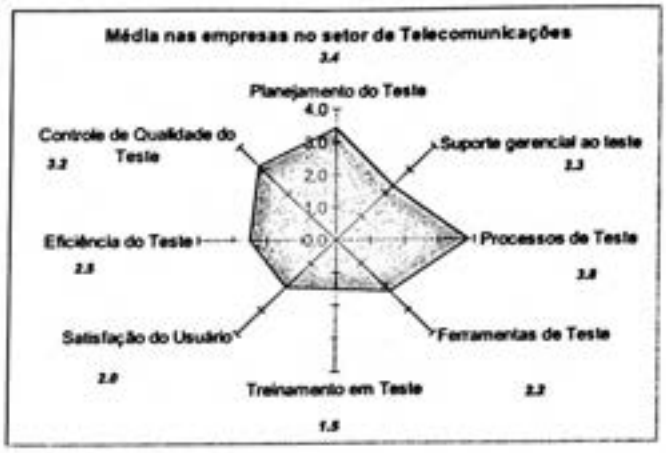

Gráfico 171 - Média nas empresas no setor de Telecomunicações

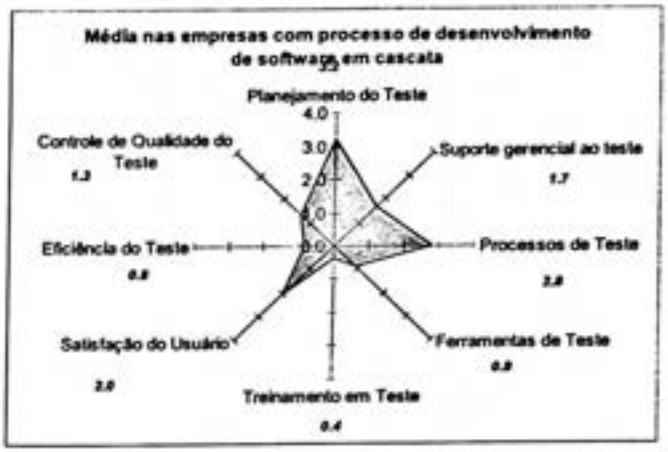

Gráfico 172 - Média nas empresas com processo de desenvolvimento de software em cascata

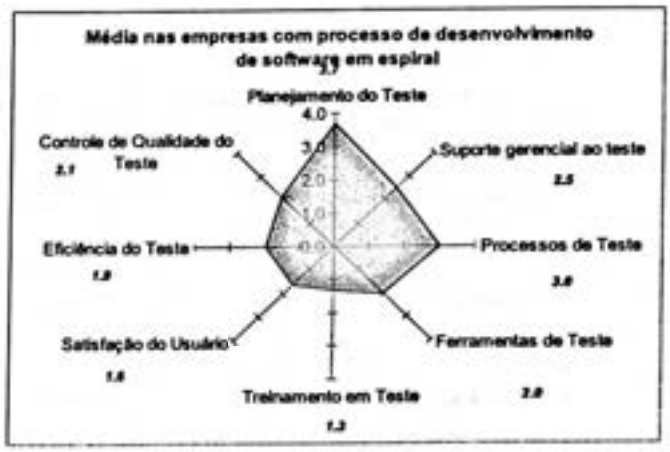

Gráfico 173 - Média nas empresas com processo de desenvolvimento de software em espiral

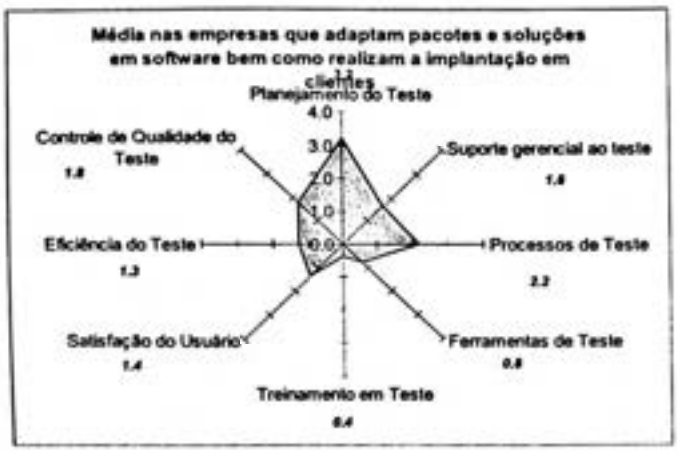

Gráfico 174 - Média nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

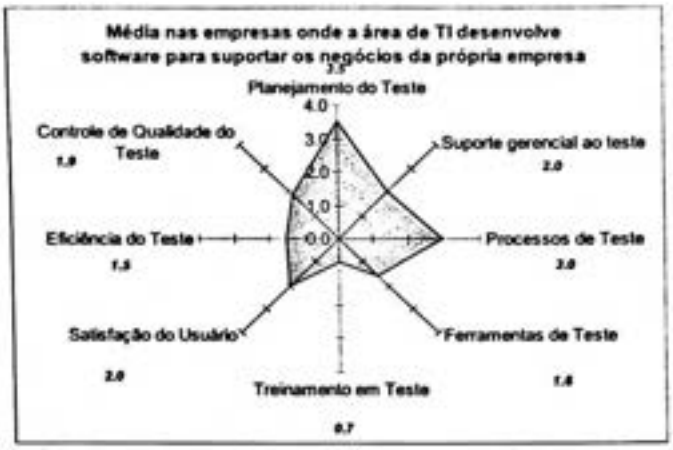

Gráfico 175 - Média nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

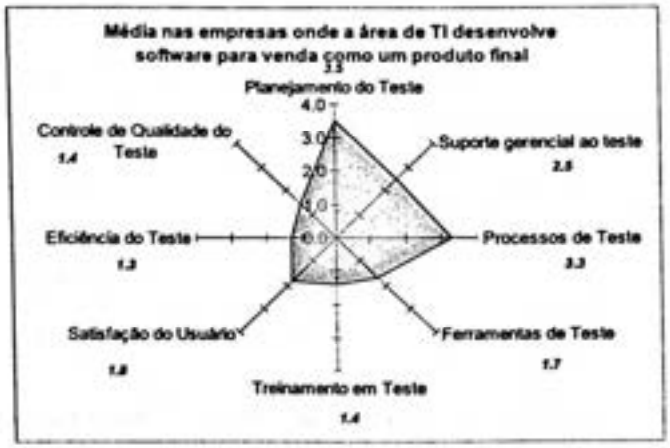

Gráfico 176 - Média nas empresas onde a área de TI desenvolve software para venda como um produto final

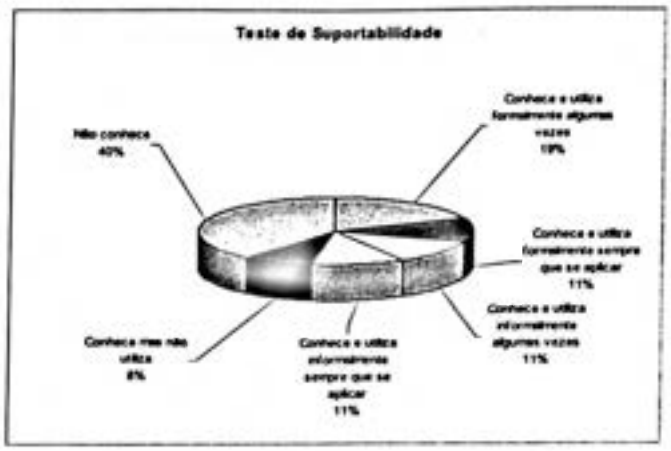

Gráfico 177 - Teste de Suportabilidade

Gráfico 178 - Teste de Suportabilidade nas empresas de grande porte

Gráfico 179 - Teste de Suportabilidade nas empresas de médio porte

Gráfico 180 - Teste de Suportabilidade nas micro empresas

Gráfico 181 - Teste de Suportabilidade nas empresas de pequeno porte

Gráfico 182 - Teste de Suportabilidade nas empresas no ramo de Energia Elétrica

Gráfico 183 - Teste de Suportabilidade nas empresas no setor Financeiro

Gráfico 184 - Teste de Suportabilidade nas empresas no setor Governamental

Gráfico 185 - Teste de Suportabilidade nas empresas no setor de Indústria e Comércio

Gráfico 186 - Teste de Suportabilidade nas empresas no setor de Prestação de Serviços

Gráfico 187 - Teste de Suportabilidade nas empresas no setor de Saúde

Gráfico 188 - Teste de Suportabilidade nas empresas no setor de Tecnologia e Informática

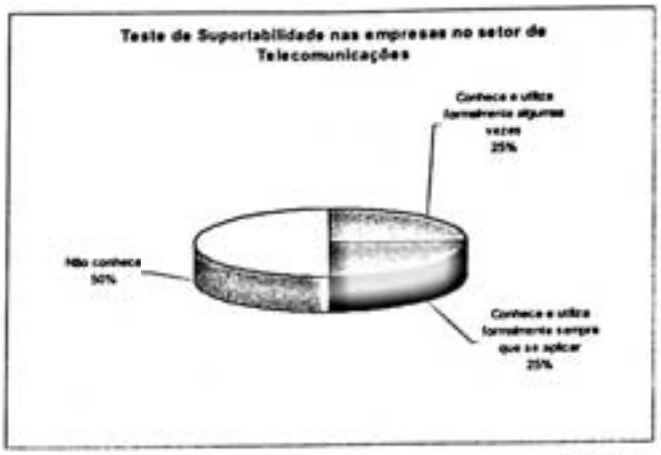

Gráfico 189 - Teste de Suportabilidade nas empresas no setor de Telecomunicações

Gráfico 190 - Teste de Suportabilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 191 - Teste de Suportabilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 192 - Teste de Suportabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 193 - Teste de Suportabilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 194 - Teste de Suportabilidade nas empresas com até 10 pessoas na área de TI

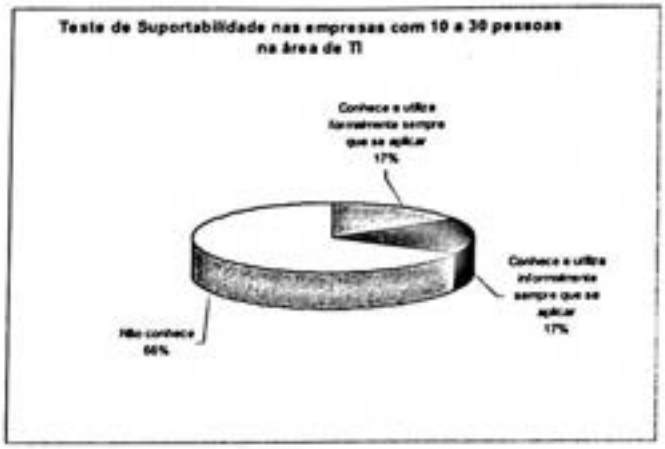

Gráfico 195 - Teste de Suportabilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 196 - Teste de Suportabilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 197 - Teste de Suportabilidade nas empresas com mais de 100 pessoas na área de TI

Gráfico 198 - Teste de Suportabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 199 - Teste de Suportabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 200 - Teste de Suportabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

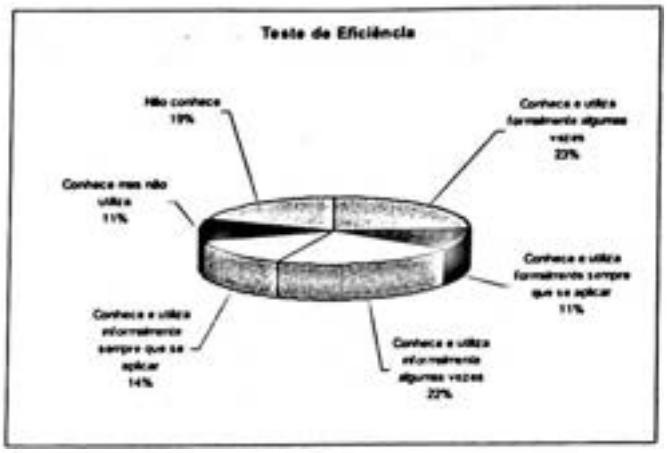

Gráfico 201 - Teste de Eficiência

Gráfico 202 - Teste de Eficiência nas empresas que realizam adapatação de pacotes/soluções de software e implantam em clientes

Gráfico 203 - Teste de Eficiência nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

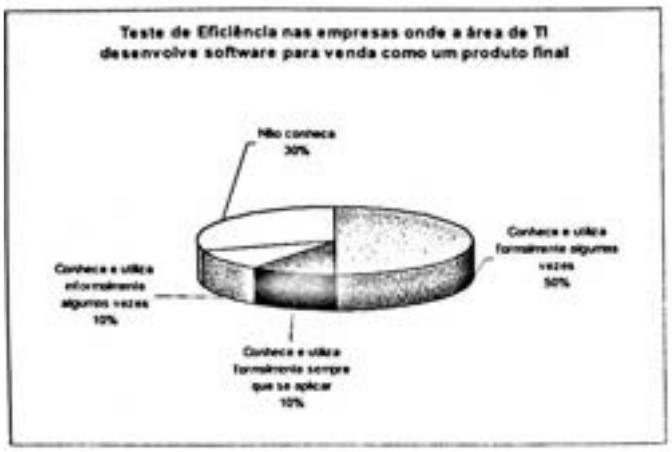

Gráfico 204 - Teste de Eficiência nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 205 - Teste de Eficiência nas empresas de grande porte

Gráfico 206 - Teste de Eficiência nas empresas de médio porte

Gráfico 207 - Teste de Eficiência nas micro empresas

Gráfico 208 - Teste de Eficiência nas empresas de pequeno porte

Gráfico 209 - Teste de Eficiência nas empresas no ramo de Energia Elétrica

Gráfico 210 - Teste de Eficiência nas empresas no setor Financeiro

Gráfico 211 - Teste de Eficiência nas empresas no setor Governamental

Gráfico 212 - Teste de Eficiência nas empresas no setor de Indústria e Comércio

Gráfico 213 - Teste de Eficiência nas empresas no setor de Prestação de Serviços

Gráfico 214 - Teste de Eficiência nas empresas no setor de Saúde

Gráfico 215 - Teste de Eficiência nas empresas no setor de Tecnologia e Informática

Gráfico 216 - Teste de Eficiência nas empresas no setor de Telecomunicações

Gráfico 217 - Teste de Eficiência nas empresas com processo de desenvolvimento de software em cascata

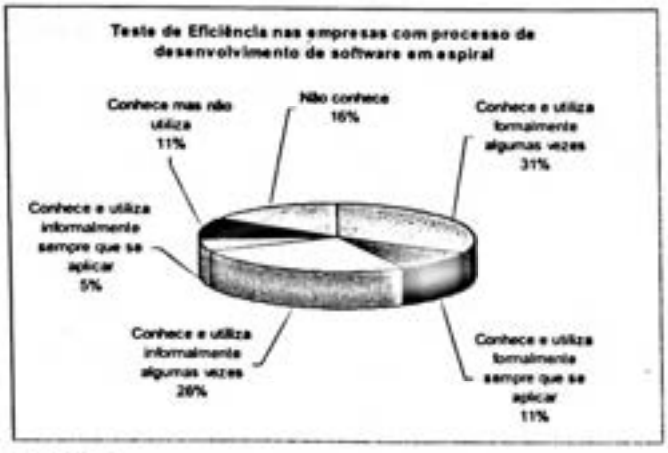

Gráfico 218 - Teste de Eficiência nas empresas com processo de desenvolvimento de software em espiral

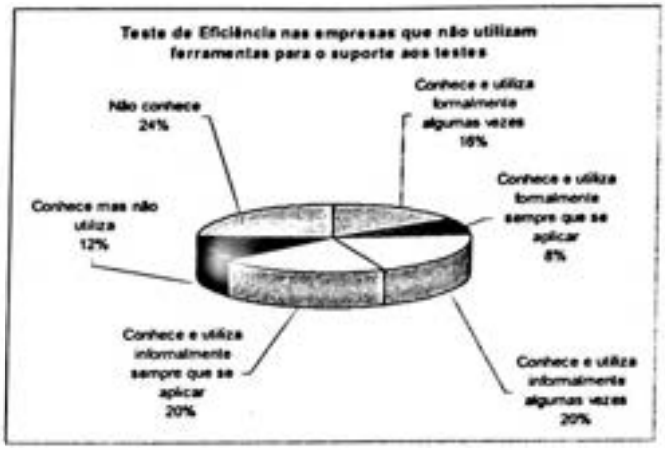

Gráfico 219 - Teste de Eficiência nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 220 - Teste de Eficiência nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 221 - Teste de Eficiência nas empresas com até 10 pessoas na área de TI

Gráfico 222 - Teste de Eficiência nas empresas com 10 a 30 pessoas na área de TI

Gráfico 223 - Teste de Eficiência nas empresas com 30 a 100 pessoas na área de TI

Gráfico 224 - Teste de Eficiência nas empresas com mais de 100 pessoas na área de TI

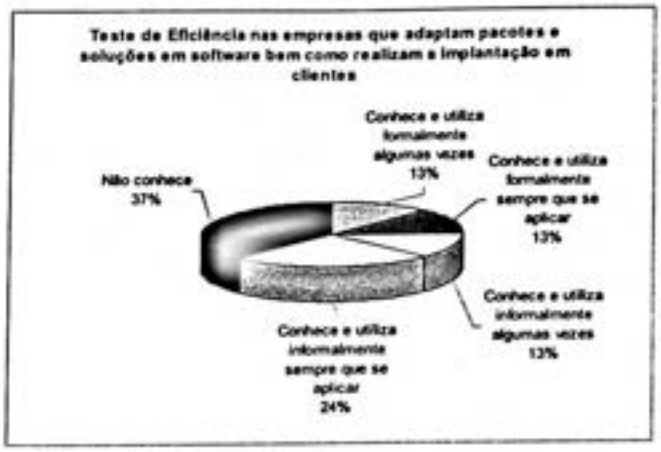

Gráfico 225 - Teste de Eficiência nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

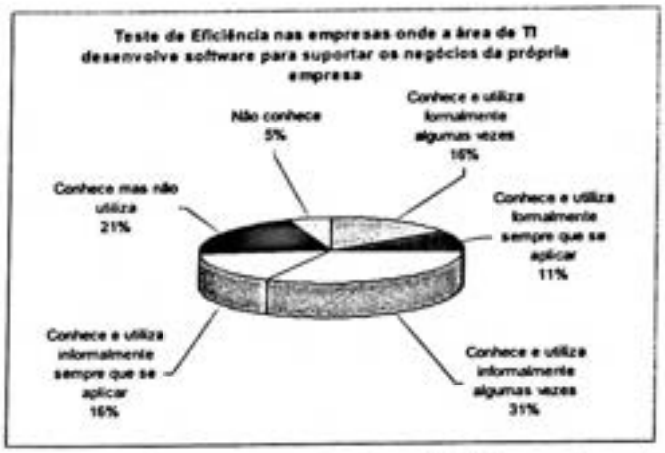

Gráfico 226 - Teste de Eficiência nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 227 - Teste de Eficiência nas empresas onde a área de TI desenvolve software para venda como um produto final

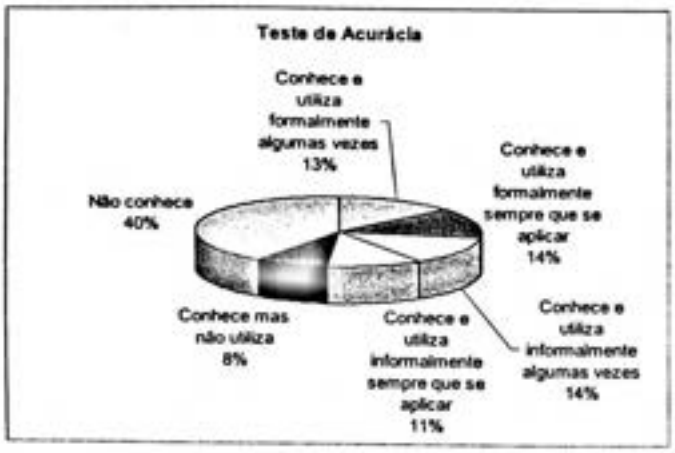

Gráfico 228 - Teste de Acurácia

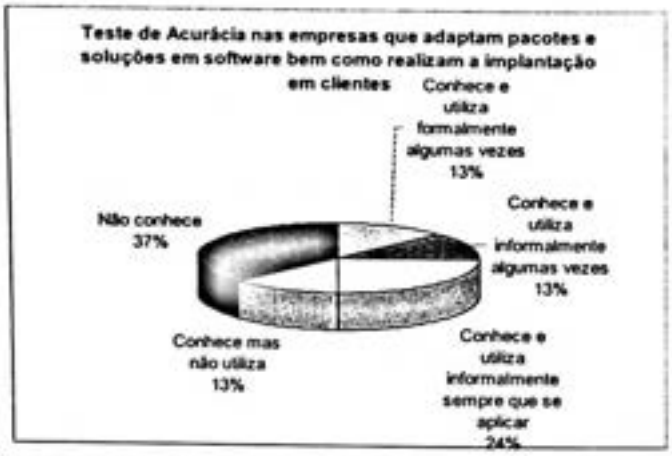

Gráfico 229 - Teste de Acurácia nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 230 - Teste de Acurácia nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

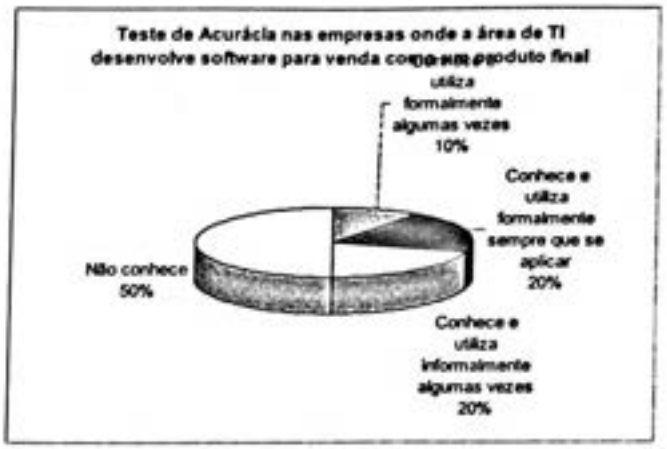

Gráfico 231 - Teste de Acurácia nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 232 - Teste de Acurácia nas empresas de grande porte

Gráfico 233 - Teste de Acurácia nas empresas de pequeno porte

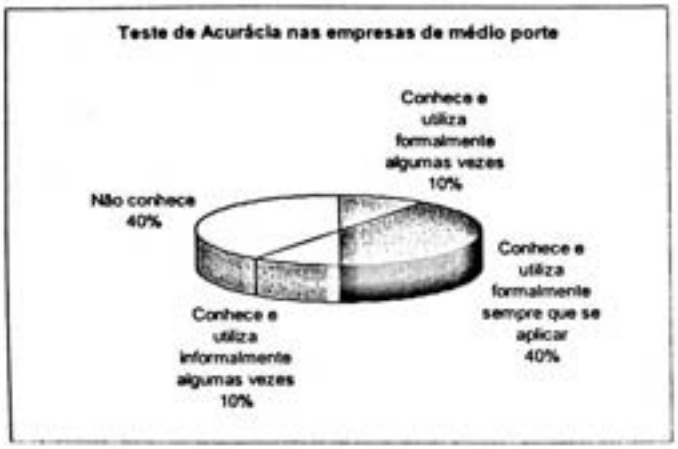

Gráfico 234 - Teste de Acurácia nas empresas de médio porte

Gráfico 235 - Teste de Acurácia nas micro empresas

Gráfico 236 - Teste de Acurácia nas empresas no ramo de Energia Elétrica

Gráfico 237 - Teste de Acurácia nas empresas no setor Financeiro

Gráfico 238 - Teste de Acurácia nas empresas no setor Governamental

Gráfico 239 - Teste de Acurácia nas empresas no setor de Indústria e Comércio

Gráfico 240 - Teste de Acurácia nas empresas no setor de Prestação de Serviços

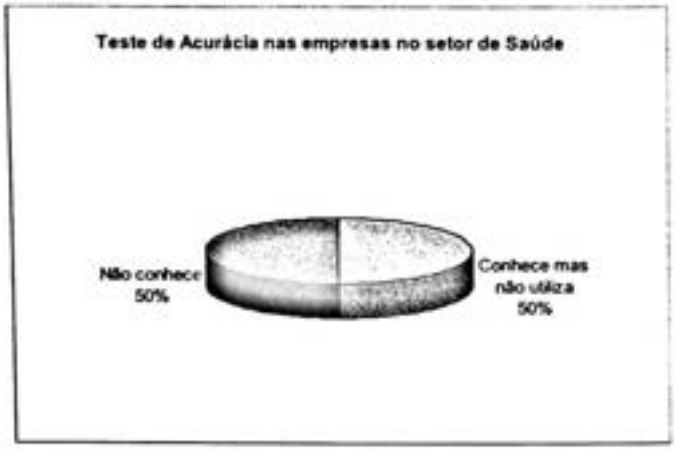

Gráfico 241 - Teste de Acurácia nas empresas no setor de Saúde

Gráfico 242 - Teste de Acurácia nas empresas no setor de Tecnologia e Informática

Gráfico 243 - Teste de Acurácia nas empresas no setor de Telecomunicações

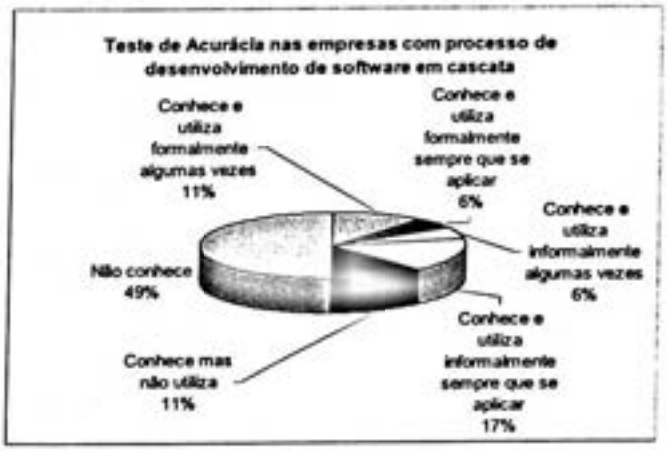

Gráfico 244 - Teste de Acurácia nas empresas com processo de desenvolvimento de software em cascata

Gráfico 245 - Teste de Acurácia nas empresas com processo de desenvolvimento de software em espiral

Gráfico 246 - Teste de Acurácia nas empresas que não utilizam ferramentas para o suporte aos testes

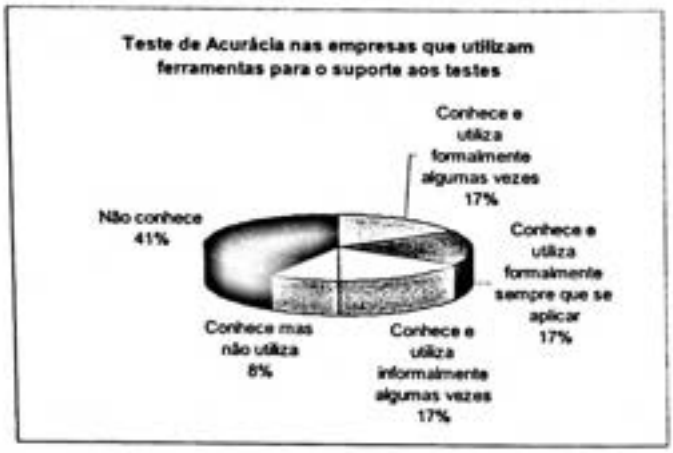

Gráfico 247 - Teste de Acurácia nas empresas que utilizam ferramentas para o suporte aos testes

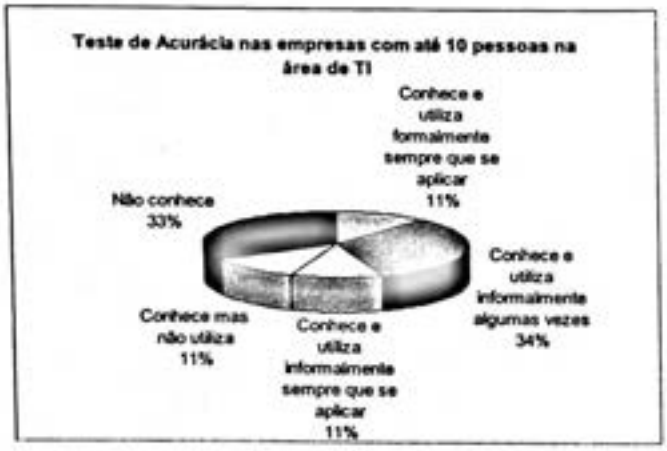

Gráfico 248 - Teste de Acurácia nas empresas com até 10 pessoas na área de TI

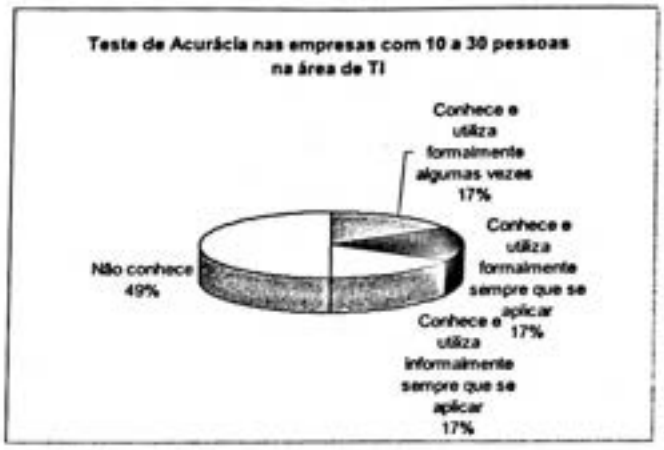

Gráfico 249 - Teste de Acurácia nas empresas com 10 a 30 pessoas na área de TI

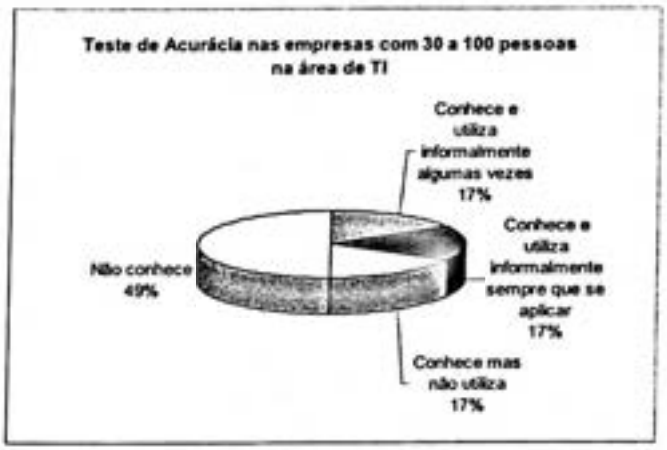

Gráfico 250 - Teste de Acurácia nas empresas com 30 a 100 pessoas na área de TI

Gráfico 251 - Teste de Acurácia nas empresas com mais de 100 pessoas na área de TI

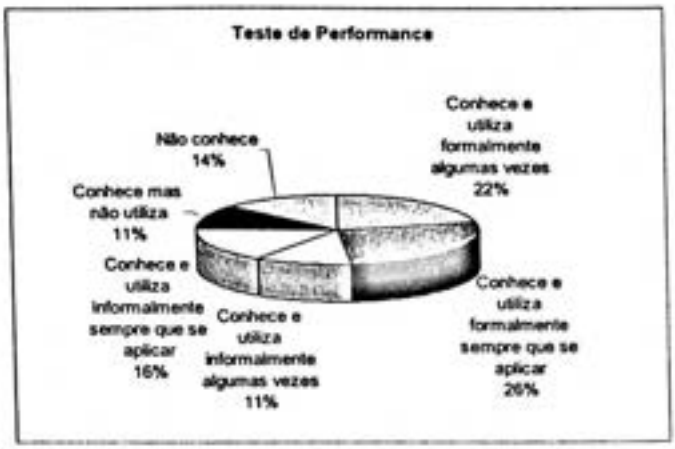

Gráfico 252 - Teste de Performance

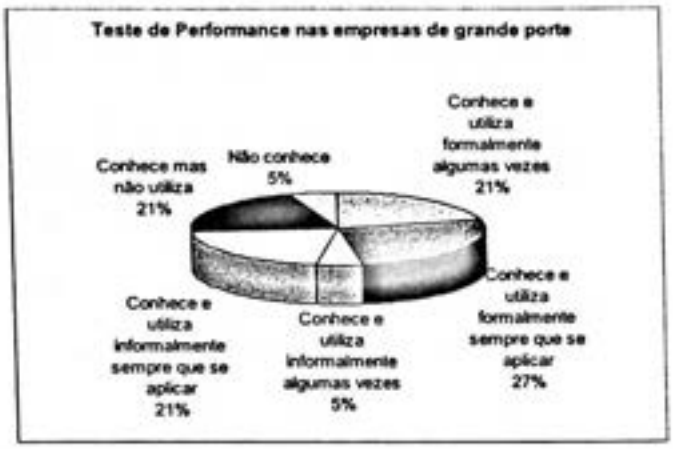

Gráfico 253 - Teste de Performance nas empresas de grande porte

Gráfico 254 - Teste de Performance nas empresas de pequeno porte

Gráfico 255 - Teste de Performance nas empresas de médio porte

Gráfico 256 - Teste de Performance nas micro empresas

Gráfico 257 - Teste de Performance nas empresas no ramo de Energia Elétrica

Gráfico 258 - Teste de Performance nas empresas no setor Financeiro

Gráfico 259 - Teste de Performance nas empresas no setor Governamental

Gráfico 260 - Teste de Performance nas empresas no setor de Indústria e Comércio

Gráfico 261 - Teste de Performance nas empresas no setor de Prestação de Serviços

Gráfico 262 - Teste de Performance nas empresas no setor de Saúde

Gráfico 263 - Teste de Performance nas empresas no setor de Tecnologia e Informática

Gráfico 264 - Teste de Performance nas empresas no setor de Telecomunicações

Gráfico 265 - Teste de Performance nas empresas com processo de desenvolvimento de software em cascata

Gráfico 266 - Teste de Performance nas empresas com processo de desenvolvimento de software em espiral

Gráfico 267 - Teste de Performance nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 268 - Teste de Performance nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 269 - Teste de Performance nas empresas com até 10 pessoas na área de TI

Gráfico 270 - Teste de Performance nas empresas com 10 a 30 pessoas na área de TI

Gráfico 271 - Teste de Performance nas empresas com 30 a 100 pessoas na área de TI

Gráfico 272 - Teste de Performance nas empresas com mais de 100 pessoas na área de TI

Gráfico 273 - Teste de Performance nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 274 - Teste de Performance nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 275 - Teste de Performance nas empresas onde a área de TI desenvolve software para venda como um produto final

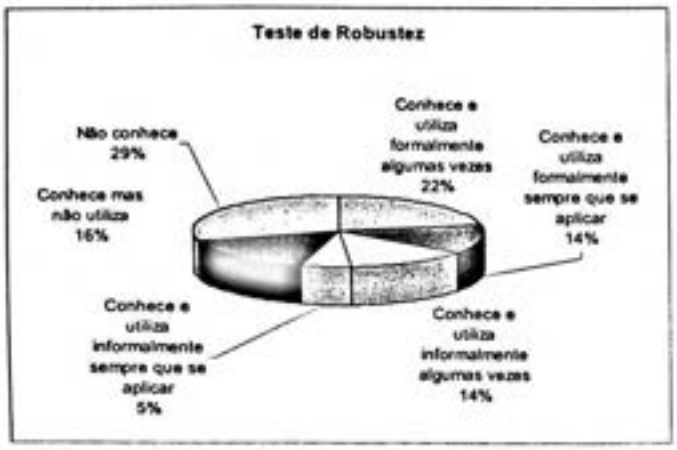

Gráfico 276 - Teste de Robustez

Gráfico 277 - Teste de Robustez nas empresas de grande porte

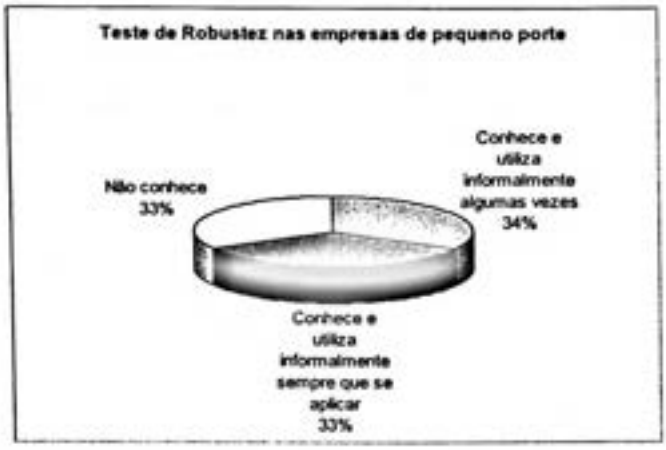

Gráfico 278 - Teste de Robustez nas empresas de pequeno porte

Gráfico 279 - Teste de Robustez nas empresas de médio porte

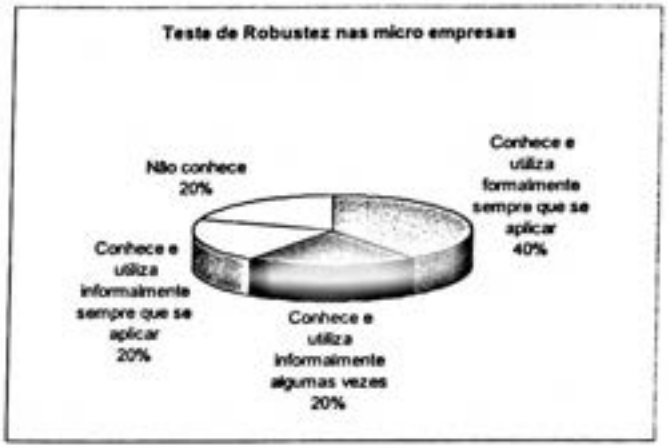

Gráfico 280 - Teste de Robustez nas micro empresas

Gráfico 281 - Teste de Robustez nas empresas no ramo de Energia Elétrica

Gráfico 282 - Teste de Robustez nas empresas no setor Financeiro

Gráfico 283 - Teste de Robustez nas empresas no setor Governamental

Gráfico 284 - Teste de Robustez nas empresas no setor de Indústria e Comércio

Gráfico 285 - Teste de Robustez nas empresas no setor de Prestação de Serviços

Gráfico 286 - Teste de Robustez nas empresas no setor de Saúde

Gráfico 287 - Teste de Robustez nas empresas no setor de Tecnologia e Informática

Gráfico 288 - Teste de Robustez nas empresas no setor de Telecomunicações

Gráfico 289 - Teste de Robustez nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 290 - Teste de Robustez nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 291 - Teste de Robustez nas empresas com até 10 pessoas na área de TI

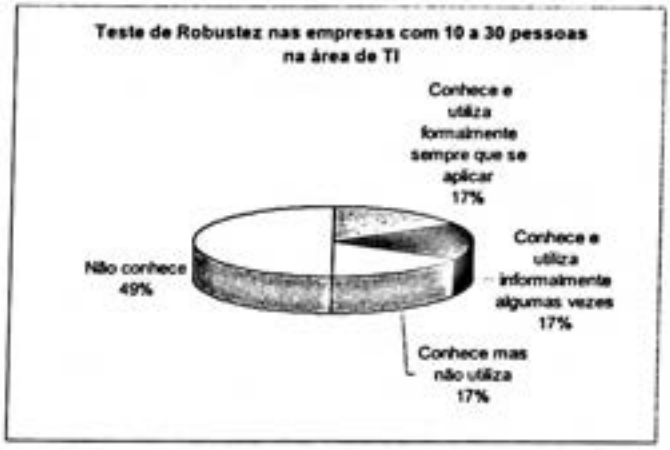

Gráfico 292 - Teste de Robustez nas empresas com 10 a 30 pessoas na área de TI

Gráfico 293 - Teste de Robustez nas empresas com 30 a 100 pessoas na área de TI

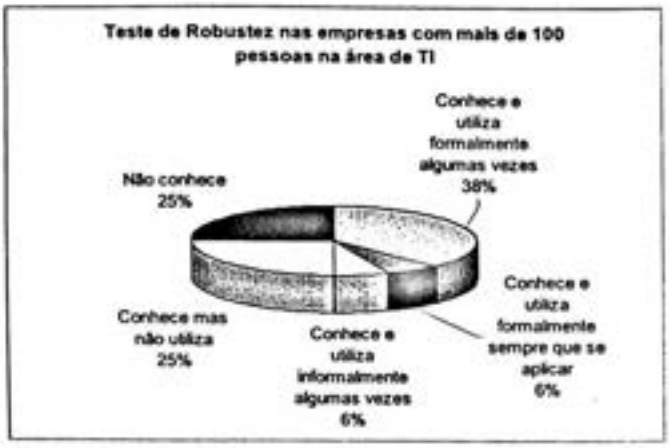

Gráfico 294 - Teste de Robustez nas empresas com mais de 100 pessoas na área de TI

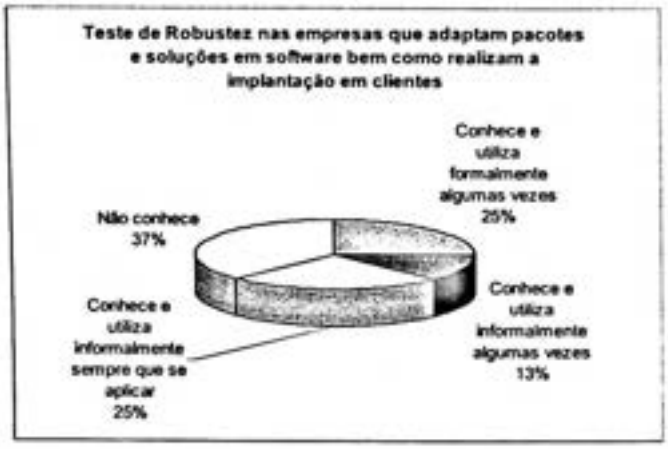

Gráfico 295 - Teste de Robustez nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

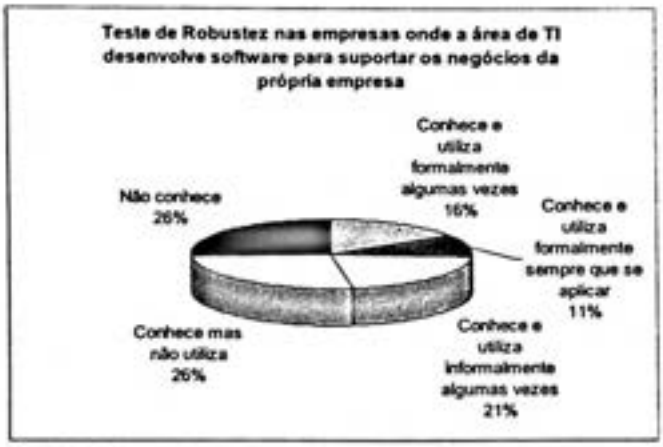

Gráfico 296 - Teste de Robustez nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

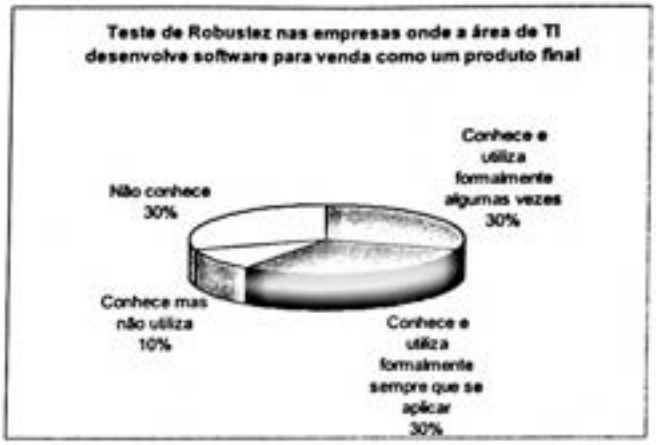

Gráfico 297 - Teste de Robustez nas empresas onde a área de TI desenvolve software para venda como um produto final

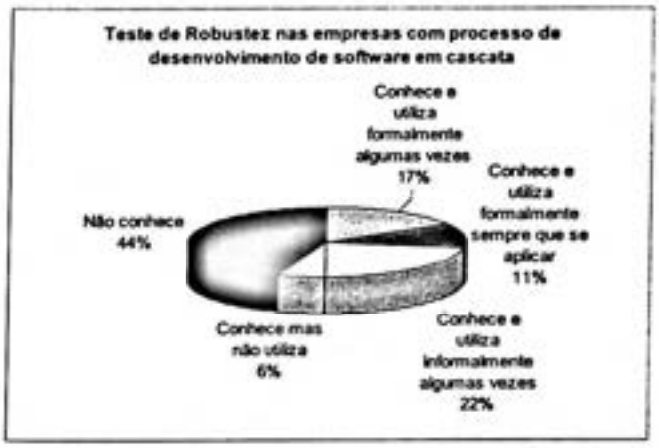

Gráfico 298 - Teste de Robustez nas empresas com processo de desenvolvimento de software em cascata

Gráfico 299 - Teste de Robustez nas empresas com processo de desenvolvimento de software em espiral

Gráfico 300 - Teste de Conformidade

Gráfico 301 - Teste de Conformidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 302 - Teste de Conformidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 303 - Teste de Conformidade nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 304 - Teste de Conformidade nas empresas de grande porte

Gráfico 305 - Teste de Conformidade nas empresas de pequeno porte

Gráfico 306 - Teste de Conformidade nas empresas de médio porte

Gráfico 307 - Teste de Conformidade nas micro empresas

Gráfico 308 - Teste de Conformidade nas empresas no ramo de Energia Elétrica

Gráfico 309 - Teste de Conformidade nas empresas no setor Financeiro

Gráfico 310 - Teste de Conformidade nas empresas no setor Governamental

Gráfico 311 - Teste de Conformidade nas empresas no setor de Indústria e Comércio

Gráfico 312 - Teste de Conformidade nas empresas no setor de Prestação de Serviços

Gráfico 313 - Teste de Conformidade nas empresas no setor de Saúde

Gráfico 314 - Teste de Conformidade nas empresas no setor de Tecnologia e Informática

Gráfico 315 - Teste de Conformidade nas empresas no setor de Telecomunicações

Gráfico 316 - Teste de Conformidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 317 - Teste de Conformidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 318 - Teste de Conformidade nas empresas com até 10 pessoas na área de TI

Gráfico 319 - Teste de Conformidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 320 - Teste de Conformidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 321 - Teste de Conformidade nas empresas com mais de 100 pessoas na área de TI

Gráfico 322 - Teste de Conformidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 323 - Teste de Conformidade nas empresas que utilizam ferramentas para o suporte aos testes

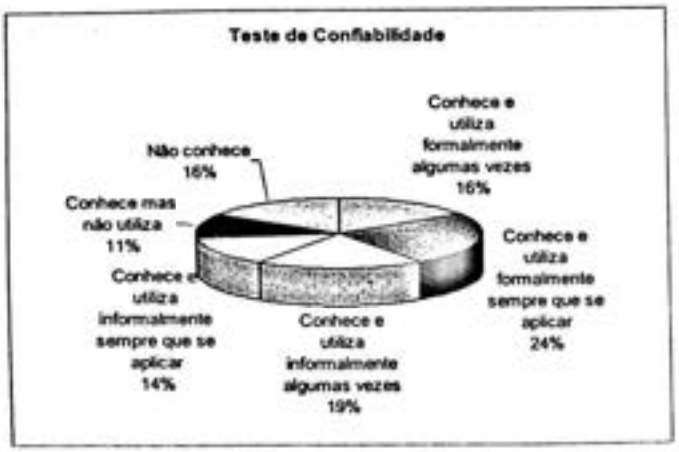

Gráfico 324 - Teste de Confiabilidade

Gráfico 325 - Teste de Confiabilidade nas empresas de grande porte

Gráfico 326 - Teste de Confiabilidade nas empresas de pequeno porte

Gráfico 327 - Teste de Confiabilidade nas empresas de médio porte

Gráfico 328 - Teste de Confiabilidade nas micro empresas

Gráfico 329 - Teste de Confiabilidade nas empresas no ramo de Energia Elétrica

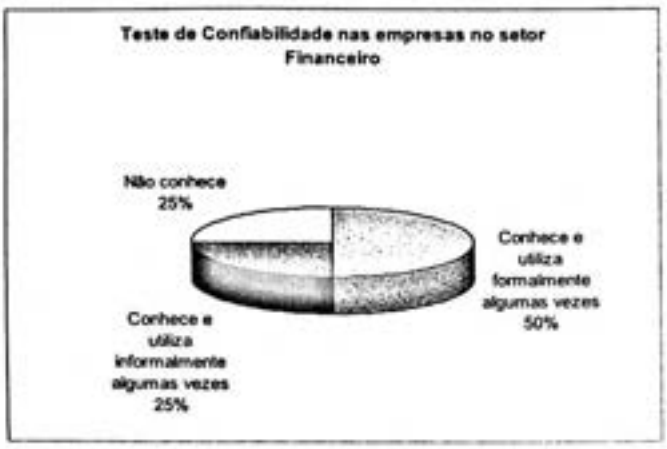

Gráfico 330 - Teste de Confiabilidade nas empresas no setor Financeiro

Gráfico 331 - Teste de Confiabilidade nas empresas no setor Governamental

Gráfico 332 - Teste de Confiabilidade nas empresas no setor de Indústria e Comércio

Gráfico 333 - Teste de Confiabilidade nas empresas no setor de Prestação de Serviços

Gráfico 334 - Teste de Confiabilidade nas empresas no setor de Saúde

Gráfico 335 - Teste de Confiabilidade nas empresas no setor de Tecnologia e Informática

Gráfico 336 - Teste de Confiabilidade nas empresas no setor de Telecomunicações

Gráfico 337 - Teste de Confiabilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 338 - Teste de Confiabilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 339 - Teste de Confiabilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 340 - Teste de Confiabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 341 - Teste de Confiabilidade nas empresas com até 10 pessoas na área de TI

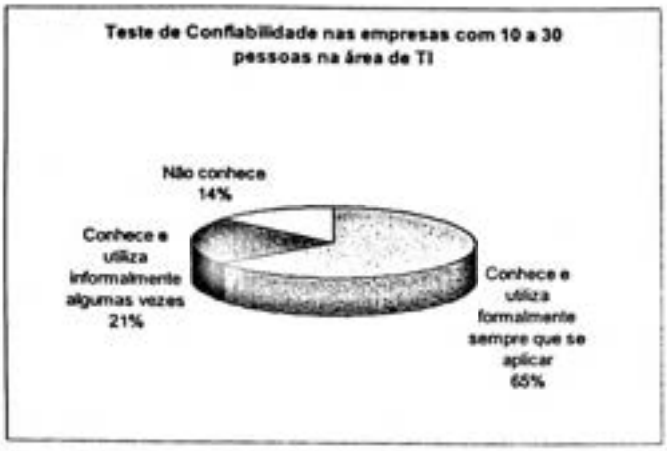

Gráfico 342 - Teste de Confiabilidade nas empresas com 10 a 30 pessoas na área de TI

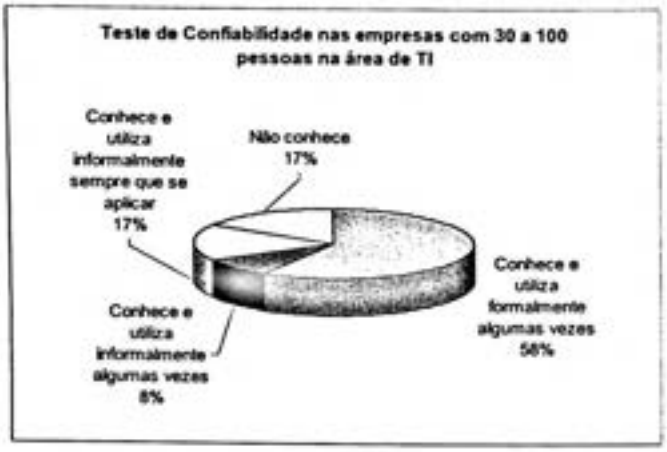

Gráfico 343 - Teste de Confiabilidade nas empresas com 30 a 100 pessoas na área de TI

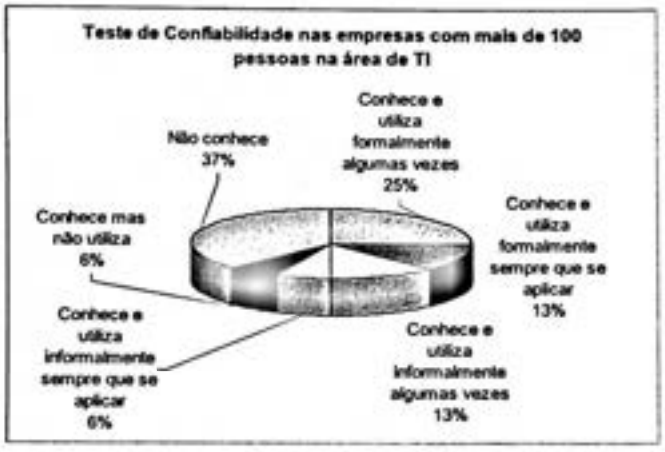

Gráfico 344 - Teste de Confiabilidade nas empresas com mais de 100 pessoas na área de TI

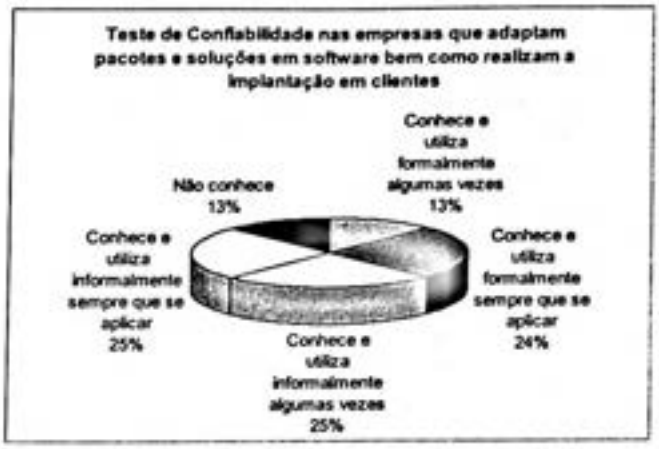

Gráfico 345 - Teste de Confiabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 346 - Teste de Confiabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 347 - Teste de Confiabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 348 - Teste de Integridade

Gráfico 349 - Teste de Integridade nas empresas de grande porte

Gráfico 350 - Teste de Integridade nas empresas de pequeno porte

Gráfico 351 - Teste de Integridade nas empresas de médio porte

Gráfico 352 - Teste de Integridade nas micro empresas

Gráfico 353 - Teste de Integridade nas empresas no ramo de Energia Elétrica

Gráfico 354 - Teste de Integridade nas empresas no setor Financeiro

Gráfico 355 - Teste de Integridade nas empresas no setor Governamental

Gráfico 356 - Teste de Integridade nas empresas no setor de Indústria e Comércio

Gráfico 357 - Teste de Integridade nas empresas no setor de Prestação de Serviços

Gráfico 358 - Teste de Integridade nas empresas no setor de Saúde

Gráfico 359 - Teste de Integridade nas empresas no setor de Tecnologia e Informática

Gráfico 360 - Teste de Integridade nas empresas no setor de Telecomunicações

Gráfico 361 - Teste de Integridade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 362 - Teste de Integridade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 363 - Teste de Integridade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 364 - Teste de Integridade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 365 - Teste de Integridade nas empresas com até 10 pessoas na área de TI

Gráfico 366 - Teste de Integridade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 367 - Teste de Integridade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 368 - Teste de Integridade nas empresas com mais de 100 pessoas na área de TI

Gráfico 369 - Teste de Integridade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 370 - Teste de Integridade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 371 - Teste de Integridade nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 372 - Teste de Usabilidade

Gráfico 373 - Teste de Usabilidade nas empresas de grande porte

Gráfico 374 - Teste de Usabilidade nas empresas de médio porte

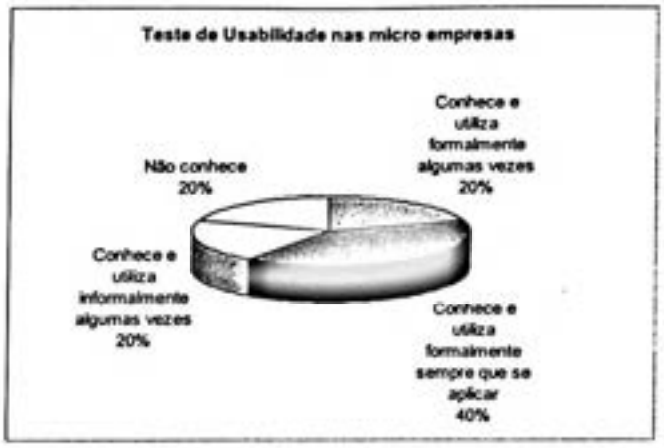

Gráfico 375 - Teste de Usabilidade nas micro empresas

Gráfico 376 - Teste de Usabilidade nas empresas de pequeno porte

Gráfico 377 - Teste de Usabilidade nas empresas no ramo de Energia Elétrica

Gráfico 378 - Teste de Usabilidade nas empresas no setor Financeiro

Gráfico 379 - Teste de Usabilidade nas empresas no setor Governamental

Gráfico 380 - Teste de Usabilidade nas empresas no setor de Indústria e Comércio

Gráfico 381 - Teste de Usabilidade nas empresas no setor de Prestação de Serviços

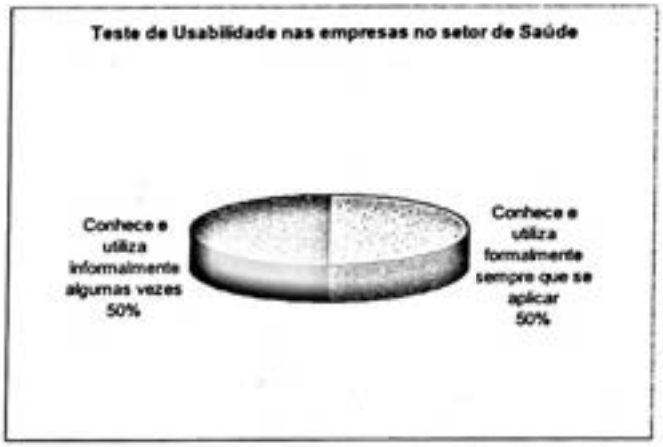

Gráfico 382 - Teste de Usabilidade nas empresas no setor de Saúde

Gráfico 383 - Teste de Usabilidade nas empresas no setor de Tecnologia e Informática

Gráfico 384 - Teste de Usabilidade nas empresas no setor de Telecomunicações

Gráfico 385 - Teste de Usabilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 386 - Teste de Usabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 387 - Teste de Usabilidade nas empresas com até 10 pessoas na área de TI

Gráfico 388 - Teste de Usabilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 389 - Teste de Usabilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 390 - Teste de Usabilidade nas empresas com mais de 100 pessoas na área de TI

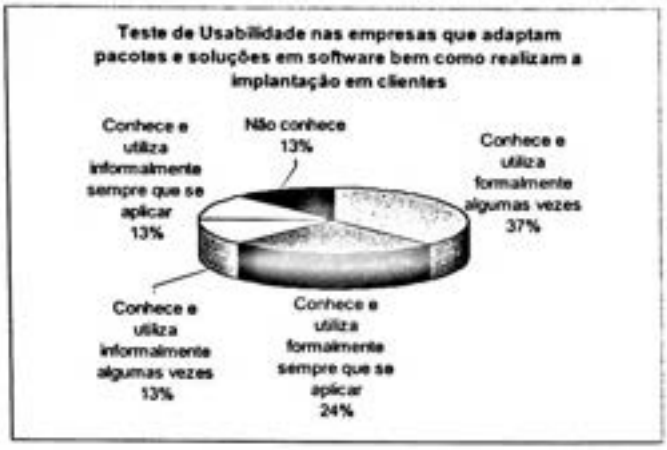

Gráfico 391 - Teste de Usabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 392 - Teste de Usabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

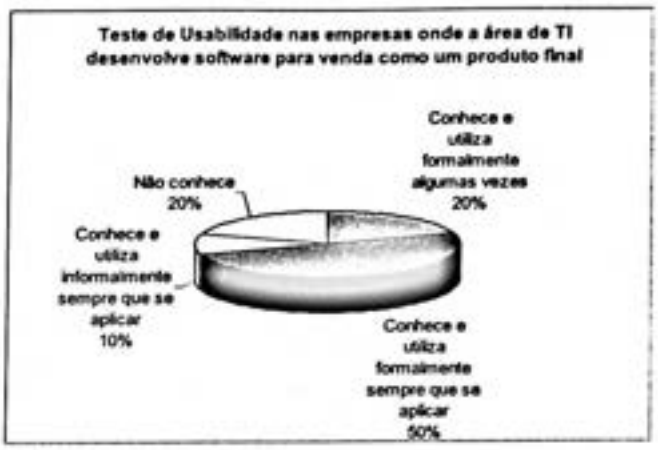

Gráfico 393 - Teste de Usabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

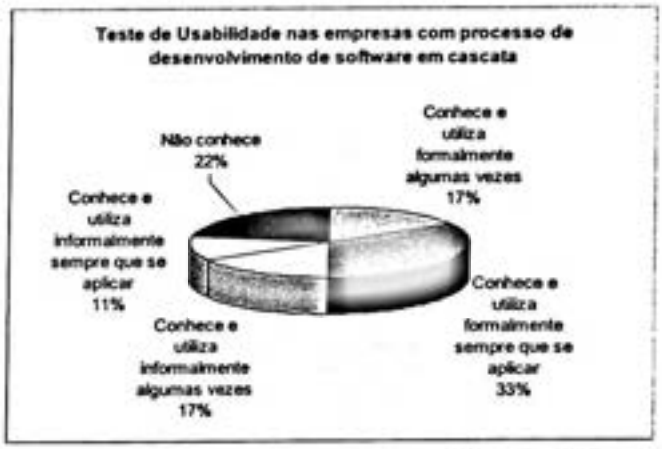

Gráfico 394 - Teste de Usabilidade nas empresas com processo de desenvolvimento de software em cascata

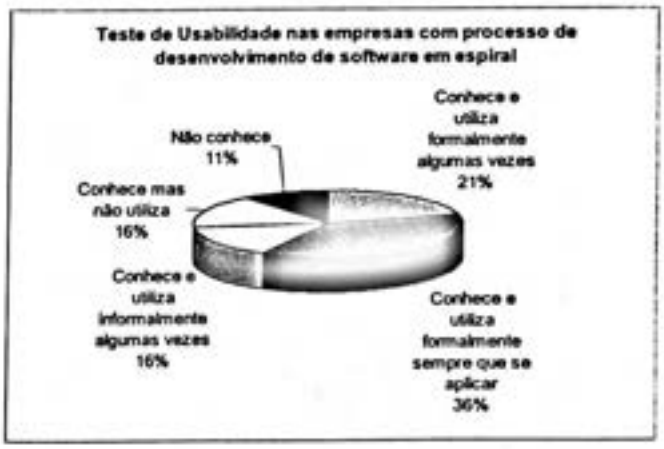

Gráfico 395 - Teste de Usabilidade nas empresas com processo de desenvolvimento de software em espiral

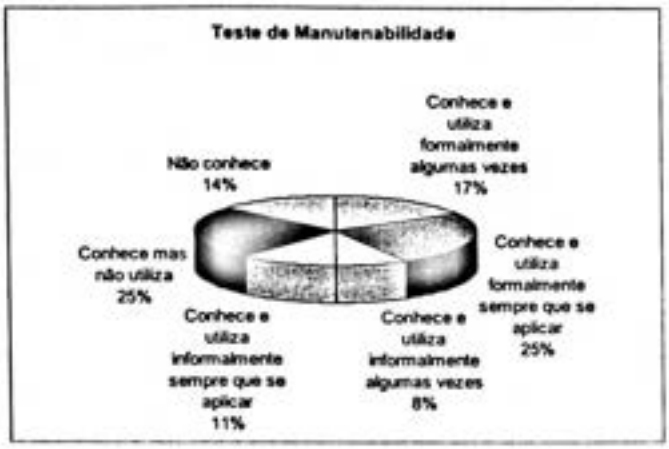

Gráfico 396 - Teste de Manutenabilidade

Gráfico 397 - Teste de Manutenabilidade nas empresas de grande porte

Gráfico 398 - Teste de Manutenabilidade nas empresas de médio porte

Gráfico 399 - Teste de Manutenabilidade nas micro empresas

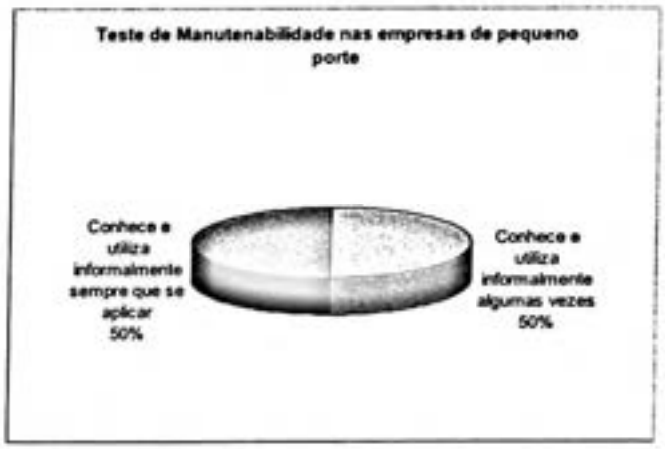

Gráfico 400 - Teste de Manutenabilidade nas empresas de pequeno porte

Gráfico 401 - Teste de Manutenabilidade nas empresas no ramo de Energia Elétrica

Gráfico 402 - Teste de Manutenabilidade nas empresas no setor Financeiro

Gráfico 403 - Teste de Manutenabilidade nas empresas no setor Governamental

Gráfico 404 - Teste de Manutenabilidade nas empresas no setor de Indústria e Comércio

Gráfico 405 - Teste de Manutenabilidade nas empresas no setor de Prestação de Serviços

Gráfico 406 - Teste de Manutenabilidade nas empresas no setor de Saúde

Gráfico 407 - Teste de Manutenabilidade nas empresas no setor de Tecnologia e Informática

Gráfico 408 - Teste de Manutenabilidade nas empresas no setor de Telecomunicações

Gráfico 409 - Teste de Manutenabilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 410 - Teste de Manutenabilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 411 - Teste de Manutenabilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 412 - Teste de Manutenabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 413 - Teste de Manutenabilidade nas empresas com até 10 pessoas na área de TI

Gráfico 414 - Teste de Manutenabilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 415 - Teste de Manutenabilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 416 - Teste de Manutenabilidade nas empresas com mais de 100 pessoas na área de TI

Gráfico 417 - Teste de Manutenabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 418 - Teste de Manutenabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 419 - Teste de Manutenabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

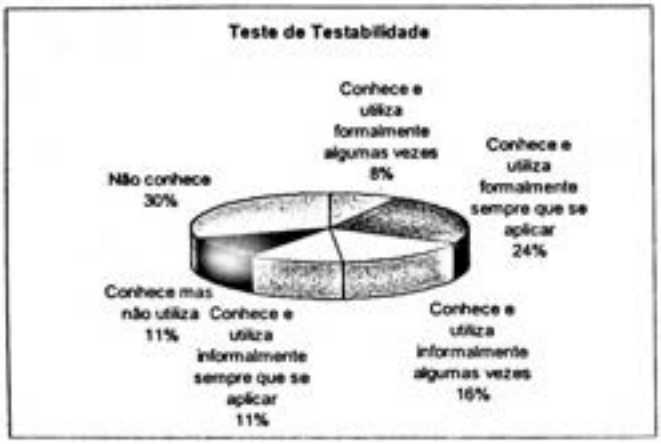

Gráfico 420 - Teste de Testabilidade

Gráfico 421 - Teste de Testabilidade nas empresas de grande porte

Gráfico 422 - Teste de Testabilidade nas empresas de médio porte

Gráfico 423 - Teste de Testabilidade nas micro empresas

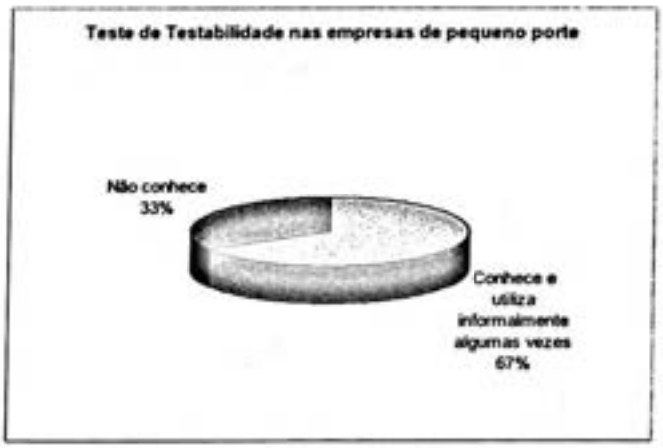

Gráfico 424 - Teste de Testabilidade nas empresas de pequeno porte

Gráfico 425 - Teste de Testabilidade nas empresas no ramo de Energia Elétrica

Gráfico 426 - Teste de Testabilidade nas empresas no setor Financeiro

Gráfico 427 - Teste de Testabilidade nas empresas no setor Governamental

Gráfico 428 - Teste de Testabilidade nas empresas no setor de Indústria e Comércio

Gráfico 429 - Teste de Testabilidade nas empresas no setor de Prestação de Serviços

Gráfico 430 - Teste de Testabilidade nas empresas no setor de Saúde

Gráfico 431 - Teste de Testabilidade nas empresas no setor de Tecnologia e Informática

Gráfico 432 - Teste de Testabilidade nas empresas no setor de Telecomunicações

Gráfico 433 - Teste de Testabilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 434 - Teste de Testabilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 435 - Teste de Testabilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 436 - Teste de Testabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 437 - Teste de Testabilidade nas empresas com até 10 pessoas na área de TI

Gráfico 438 - Teste de Testabilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 439 - Teste de Testabilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 440 - Teste de Testabilidade nas empresas com mais de 100 pessoas na área de TI

Gráfico 441 - Teste de Testabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 442 - Teste de Testabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 443 - Teste de Testabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 444 - Teste de Flexibilidade

Gráfico 445 - Teste de Flexibilidade nas empresas de grande porte

Gráfico 446 - Teste de Flexibilidade nas empresas de médio porte

Gráfico 447 - Teste de Flexibilidade nas micro empresas

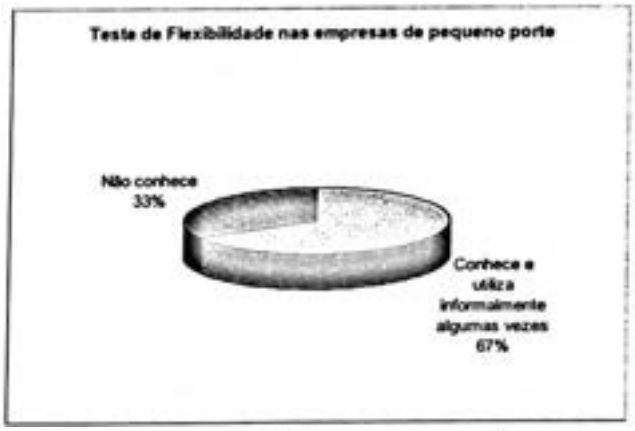

Gráfico 448 - Teste de Flexibilidade nas empresas de pequeno porte

Gráfico 449 - Teste de Flexibilidade nas empresas no ramo de Energia Elétrica

Gráfico 450 - Teste de Flexibilidade nas empresas no setor Financeiro

Gráfico 451 - Teste de Flexibilidade nas empresas no setor Governamental

Gráfico 452 - Teste de Flexibilidade nas empresas no setor de Indústria e Comércio

Gráfico 453 - Teste de Flexibilidade nas empresas no setor de Prestação de Serviços

Gráfico 454 - Teste de Flexibilidade nas empresas no setor de Saúde

Gráfico 455 - Teste de Flexibilidade nas empresas no setor de Tecnologia e Informática

Gráfico 456 - Teste de Flexibilidade nas empresas no setor de Telecomunicações

Gráfico 457 - Teste de Flexibilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 458 - Teste de Flexibilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 459 - Teste de Flexibilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 460 - Teste de Flexibilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 461 - Teste de Flexibilidade nas empresas com até 10 pessoas na área de TI

Gráfico 462 - Teste de Flexibilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 463 - Teste de Flexibilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 464 - Teste de Flexibilidade nas empresas com mais de 100 pessoas na área de TI

Gráfico 465 - Teste de Flexibilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 466 - Teste de Flexibilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 467 - Teste de Flexibilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

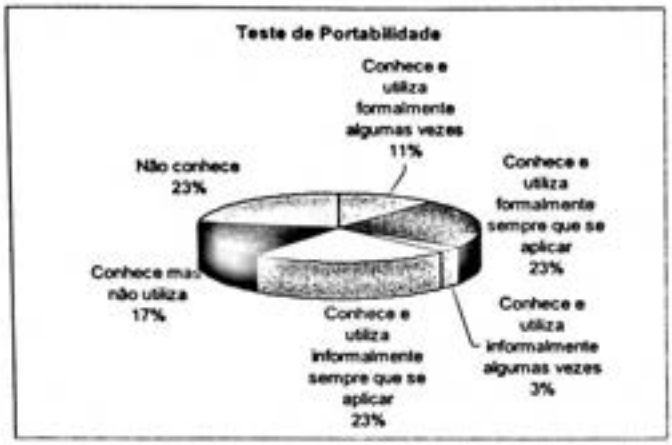

Gráfico 468 - Teste de Portabilidade

Gráfico 469 - Teste de Portabilidade nas empresas de grande porte

Gráfico 470 - Teste de Portabilidade nas empresas de médio porte

Gráfico 471 - Teste de Portabilidade nas micro empresas

Gráfico 472 - Teste de Portabilidade nas empresas de pequeno porte

Gráfico 473 - Teste de Portabilidade nas empresas no ramo de Energia Elétrica

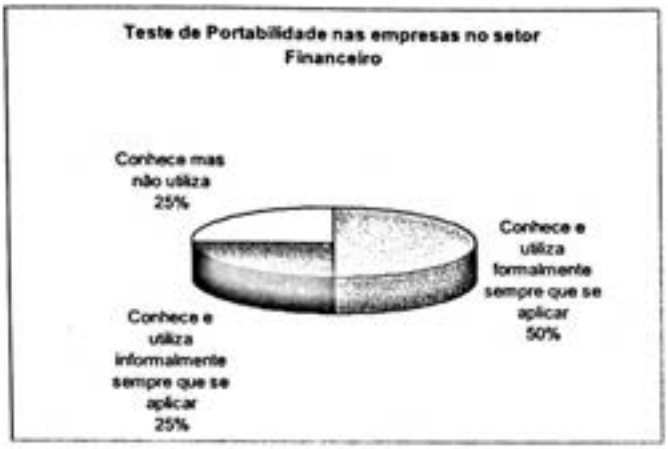

Gráfico 474 - Teste de Portabilidade nas empresas no setor Financeiro

Gráfico 475 - Teste de Portabilidade nas empresas no setor Governamental

Gráfico 476 - Teste de Portabilidade nas empresas no setor de Indústria e Comércio

Gráfico 477 - Teste de Portabilidade nas empresas no setor de Prestação de Serviços

Gráfico 478 - Teste de Portabilidade nas empresas no setor de Saúde

Gráfico 479 - Teste de Portabilidade nas empresas no setor de Tecnologia e Informática

Gráfico 480 - Teste de Portabilidade nas empresas no setor de Telecomunicações

Gráfico 481 - Teste de Portabilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 482 - Teste de Portabilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 483 - Teste de Portabilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 484 - Teste de Portabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 485 - Teste de Portabilidade nas empresas com até 10 pessoas na área de TI

Gráfico 486 - Teste de Portabilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 487 - Teste de Portabilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 488 - Teste de Portabilidade nas empresas com mais de 100 pessoas na área de TI

Gráfico 489 - Teste de Portabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 490 - Teste de Portabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 491 - Teste de Portabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

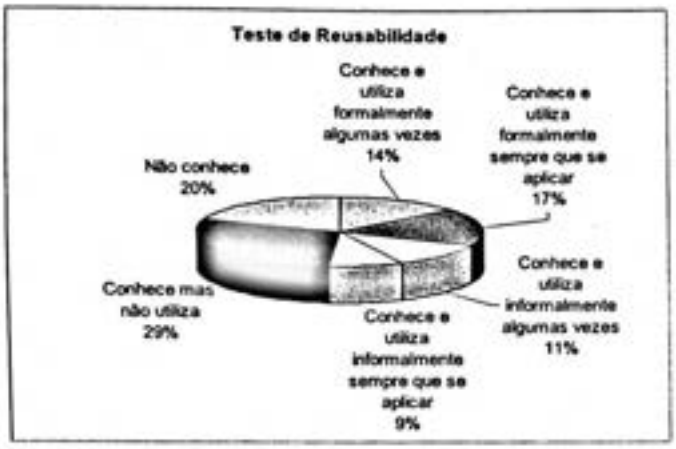

Gráfico 492 - Teste de Reusabilidade

Gráfico 493 - Teste de Reusabilidade nas empresas no setor de Prestação de Serviços

Gráfico 494 - Teste de Reusabilidade nas empresas no setor de Saúde

Gráfico 495 - Teste de Reusabilidade nas empresas no setor de Tecnologia e Informática

Gráfico 496 - Teste de Reusabilidade nas empresas no setor de Telecomunicações

Gráfico 497 - Teste de Reusabilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 498 - Teste de Reusabilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 499 - Teste de Reusabilidade nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 500 - Teste de Reusabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 501 - Teste de Reusabilidade nas empresas com até 10 pessoas na área de TI

Gráfico 502 - Teste de Reusabilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 503 - Teste de Reusabilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 504 - Teste de Reusabilidade nas empresas com mais de 100 pessoas na área de TI

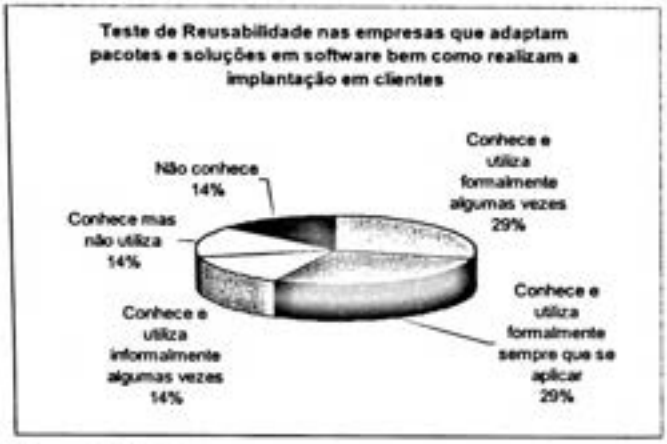

Gráfico 505 - Teste de Reusabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 506 - Teste de Reusabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 507 - Teste de Reusabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 508 - Teste de Reusabilidade nas empresas de grande porte

Gráfico 509 - Teste de Reusabilidade nas empresas de médio porte

Gráfico 510 - Teste de Reusabilidade nas micro empresas

Gráfico 511 - Teste de Reusabilidade nas empresas de pequeno porte

Gráfico 512 - Teste de Reusabilidade nas empresas no ramo de Energia Elétrica

Gráfico 513 - Teste de Reusabilidade nas empresas no setor Financeiro

Gráfico 514 - Teste de Reusabilidade nas empresas no setor Governamental

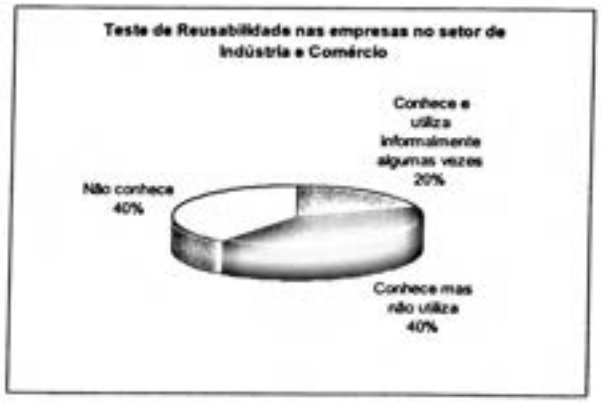

Gráfico 515 - Teste de Reusabilidade nas empresas no setor de Indústria e Comércio

Gráfico 516 - Teste de Interoperabilidade

Gráfico 517 - Teste de Interoperabilidade nas empresas de grande porte

Gráfico 518 - Teste de Interoperabilidade nas empresas de médio porte

Gráfico 519 - Teste de Interoperabilidade nas micro empresas

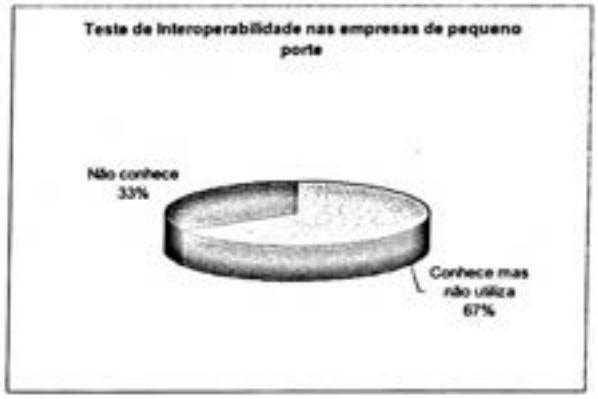

Gráfico 520 - Teste de Interoperabilidade nas empresas de pequeno porte

Gráfico 521 - Teste de Interoperabilidade nas empresas no ramo de Energia Elétrica

Gráfico 522 - Teste de Interoperabilidade nas empresas no setor Financeiro

Gráfico 523 - Teste de Interoperabilidade nas empresas no setor Governamental

Gráfico 524 - Teste de Interoperabilidade nas empresas no setor de Indústria e Comércio

Gráfico 525 - Teste de Interoperabilidade nas empresas no setor de Prestação de Serviços

Gráfico 526 - Teste de Interoperabilidade nas empresas no setor de Saúde

Gráfico 527 - Teste de Interoperabilidade nas empresas no setor de Tecnologia e Informática

Gráfico 528 - Teste de Interoperabilidade nas empresas no setor de Telecomunicações

Gráfico 529 - Teste de Interoperabilidade nas empresas que não utilizam ferramentas para o suporte aos testesrabilidade

Gráfico 530 - Teste de Interoperabilidade nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 531 - Teste de Interoperabilidade nas empresas com processo de desenvolvimento de software em cascata

Gráfico 532 - Teste de Interoperabilidade nas empresas com processo de desenvolvimento de software em espiral

Gráfico 533 - Teste de Interoperabilidade nas empresas com até 10 pessoas na área de TI

Gráfico 534 - Teste de Interoperabilidade nas empresas com 10 a 30 pessoas na área de TI

Gráfico 535 - Teste de Interoperabilidade nas empresas com 30 a 100 pessoas na área de TI

Gráfico 536 - Teste de Interoperabilidade nas empresas com mais de 100 pessoas na área de TI

Gráfico 537 - Teste de Interoperabilidade nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 538 - Teste de Interoperabilidade nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 539 - Teste de Interoperabilidade nas empresas onde a área de TI desenvolve software para venda como um produto final

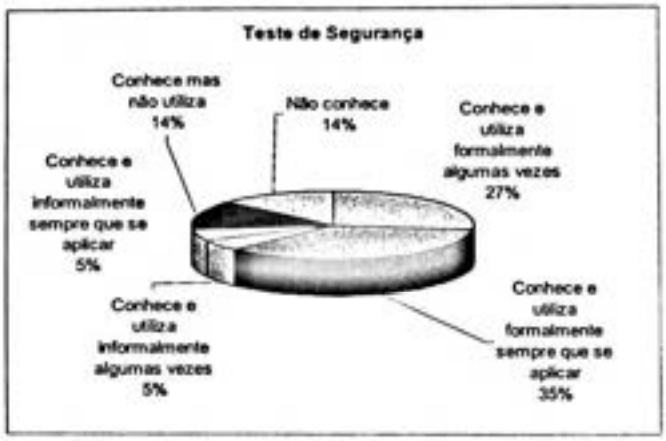

Gráfico 540 - Teste de Segurança

Gráfico 541 - Teste de Segurança nas empresas de grande porte

Gráfico 542 - Teste de Segurança nas empresas de médio porte

Gráfico 543 - Teste de Segurança nas micro empresas

Gráfico 544 - Teste de Segurança nas empresas de pequeno porte

Gráfico 545 - Teste de Segurança nas empresas no ramo de Energia Elétrica

Gráfico 546 - Teste de Segurança nas empresas no setor Financeiro

Gráfico 547 - Teste de Segurança nas empresas no setor Governamental

Gráfico 548 - Teste de Segurança nas empresas no setor de Indústria e Comércio

Gráfico 549 - Teste de Segurança nas empresas no setor de Prestação de Serviços

Gráfico 550 - Teste de Segurança nas empresas no setor de Saúde

Gráfico 551 - Teste de Segurança nas empresas no setor de Tecnologia e Informática

Gráfico 552 - Teste de Segurança nas empresas no setor de Telecomunicações

Gráfico 553 - Teste de Segurança nas empresas com processo de desenvolvimento de software em cascata

Gráfico 554 - Teste de Segurança nas empresas com processo de desenvolvimento de software em espiral

Gráfico 555 - Teste de Segurança nas empresas que não utilizam ferramentas para o suporte aos testes

Gráfico 556 - Teste de Segurança nas empresas que utilizam ferramentas para o suporte aos testes

Gráfico 557 - Teste de Segurança nas empresas com até 10 pessoas na área de TI

Gráfico 558 - Teste de Segurança nas empresas com 10 a 30 pessoas na área de TI

Gráfico 559 - Teste de Segurança nas empresas com 30 a 100 pessoas na área de TI

Gráfico 560 - Teste de Segurança nas empresas com mais de 100 pessoas na área de TI

Gráfico 561 - Teste de Segurança nas empresas que adaptam pacotes e soluções em software bem como realizam a implantação em clientes

Gráfico 562 - Teste de Segurança nas empresas onde a área de TI desenvolve software para suportar os negócios da própria empresa

Gráfico 563 - Teste de Segurança nas empresas onde a área de TI desenvolve software para venda como um produto final

Gráfico 564 - Atributos de Qualidade - Média Geral

Gráfico 565 - Atributos de Qualidade em empresas de Médio Porte

Gráfico 566 - Atributos de Qualidade em empresas de Grande Porte

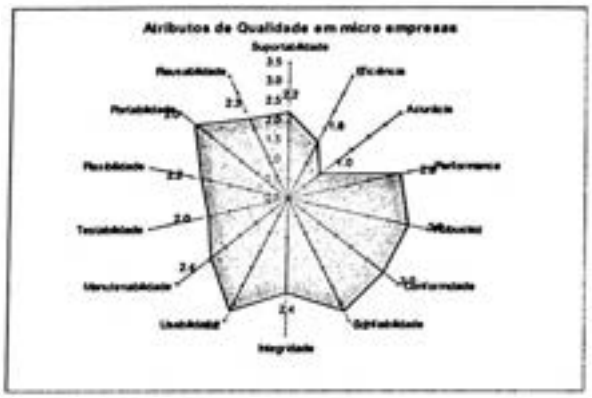

Gráfico 567 - Atributos de Qualidade em micro empresas

Gráfico 568 - Atributos de Qualidade em empresas de Pequeno Porte

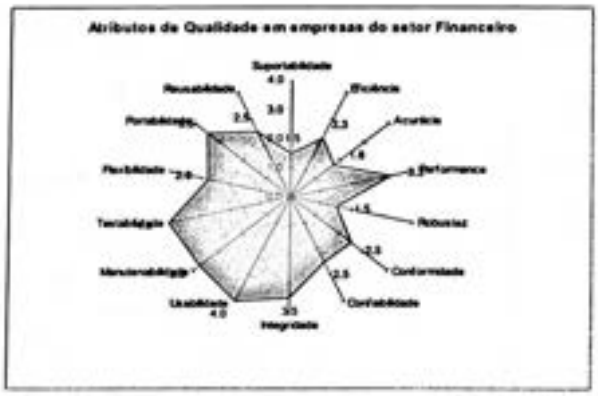

Gráfico 569 - Atributos de Qualidade em empresas do setor Financeiro

Gráfico 570 - Atributos de Qualidade em empresas do setor da Indústria e Comércio

Gráfico 571 - Atributos de Qualidade em empresas do setor da Prestação de Serviços

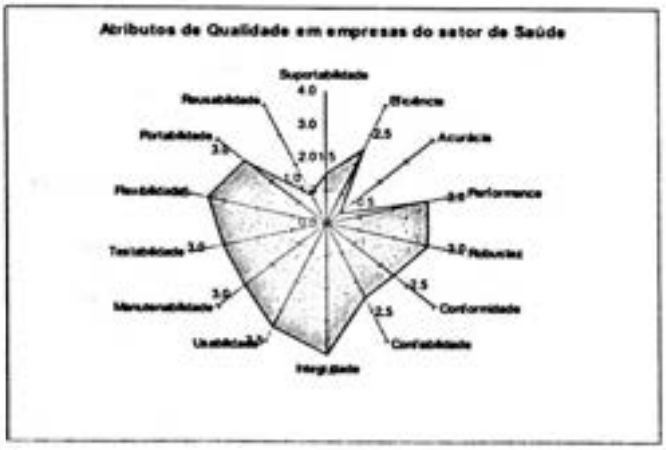

Gráfico 572 - Atributos de Qualidade em empresas do setor de Saúde

Gráfico 573 - Atributos de Qualidade em empresas do setor de Tecnologia da Informação

Gráfico 574 - Atributos de Qualidade em empresas do setor de Telecomunicações

Gráfico 575 - Atributos de Qualidade em empresas com processo de desenvolvimento de software em cascata

Gráfico 576 - Atributos de Qualidade em empresas com processo de desenvolvimento de software em espiral

Gráfico 577 - Atributos de Qualidade em empresas com Área de TI com até 10 pessoas

Gráfico 578 - Atributos de Qualidade em empresas com Área de TI de 10 a 30 pessoas

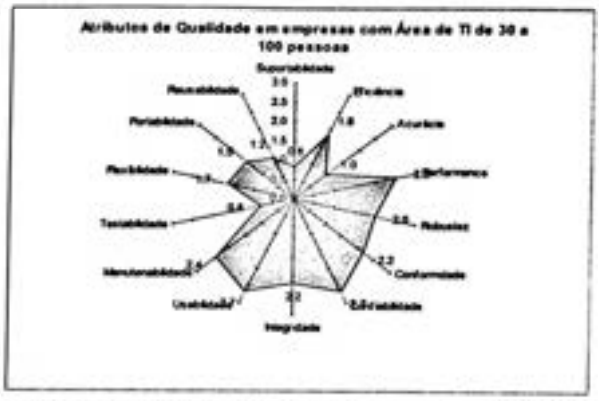

Gráfico 579 - Atributos de Qualidade em empresas com Área de TI de 30 a 100 pessoas

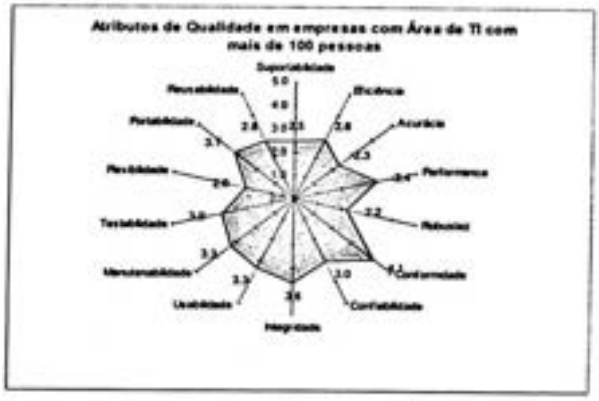

Gráfico 580 - Atributos de Qualidade em empresas com Área de TI com mais de 100 pessoas

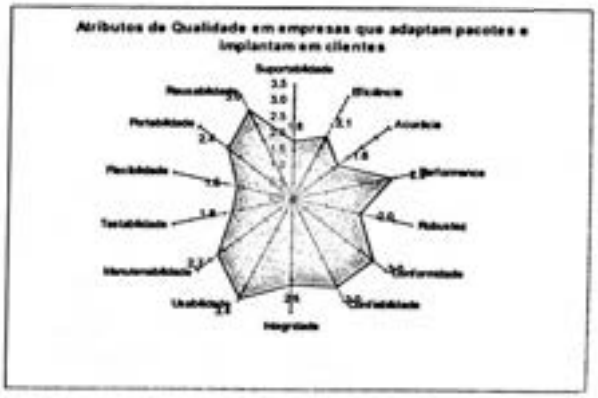

Gráfico 581 - Atributos de Qualidade em empresas que adaptam pacotes e implantam em clientes

Gráfico 582 - Atributos de Qualidade em empresas que desenvolvem software para suportar as necessidades de sua área de negócios

Gráfico 583 - Atributos de Qualidade em empresas com desenvolvimento para a venda do software como um produto final

Gráfico 584 - Atributos de Qualidade em empresas onde os usuários tem um nível de satisfação muito alto, na opinião dos respondentes

Gráfico 585 - Atributos de Qualidade em empresas onde os usuários tem um nível de satisfação alto, na opinião dos respondentes

Gráfico 586 - Atributos de Qualidade em empresas onde os usuários tem um nível de satisfação médio, na opinião dos respondentes

Gráfico 587 - Atributos de Qualidade em empresas onde os usuários tem um nível de satisfação baixo, na opinião dos respondentes