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

SIMONE MOREIRA CUNHA

ENGENHARIA DE *SOFTWARE* – UMA ABORDAGEM À FASE DE TESTES

Belo Horizonte
2010

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação
Especialização em Informática: Ênfase: Análise de Sistemas

ENGENHARIA DE *SOFTWARE* – UMA ABORDAGEM À FASE DE TESTES

por

SIMONE MOREIRA CUNHA

Monografia de Final de Curso

Prof. *Ângelo Moura Guimarães*

Belo Horizonte
2010

SIMONE MOREIRA CUNHA

ENGENHARIA DE *SOFTWARE* – UMA ABORDAGEM À FASE DE TESTES

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciências Exatas da Universidade Federal de Minas Gerais, como atividade da disciplina Engenharia de *Software*, ministrada pelo professor Ângelo Moura Guimarães.

Área de concentração: Análise de Sistemas

Orientador(a): Prof. *Ângelo Moura Guimarães.*

Belo Horizonte
2010

RESUMO

O assunto a ser tratado no decorrer da pesquisa será Engenharia de Software, tendo como área de concentração Testes de Software. A fase de Testes, dentro da Engenharia de Software, trata da investigação do software, com o objetivo de fornecer informações sobre sua qualidade em relação ao contexto em que ele deve operar. A proposta centralizadora da pesquisa é investigar a fase de Testes apresentando: conceitos, fluxo de trabalho, atividades, artefatos e diretrizes. Além disso, objetiva-se conhecer padrões importantes, tais como PMI, QAI e IEEE 829, de forma a assegurar a qualidade no desenvolvimento da documentação da fase de testes. Assim, neste trabalho são apresentadas as Estratégias de Teste, abordando o processo V&V (Validação e Verificação), as Técnicas de Teste de Software, abordando os projetos de casos de teste bem como padrões de teste. As fases de teste (estratégias e técnicas de teste) são estudadas e aplicadas na avaliação de um projeto real. Apesar de algumas restrições por parte do setor de desenvolvimento em trabalhar com a documentação proposta nesta fase, concluísse que, com a adoção das estratégias e técnicas, é possível obter: qualidade de *software*, aplicativos finais sendo executados com menor número de erros possíveis, facilidade na operação de manutenção de *software* e controle sobre desenvolvimento dentro de custos, prazos e níveis de qualidade desejados.

Palavras-chave: Estratégias de Teste, Técnicas de Teste, V&V

ABSTRACT

During this work, we discuss Software Engineering, focusing in Software Test area. Software test deals with the software investigation, providing information about their quality in relation to the context in which it must operate.
The main goal of the research is to investigate the Test Phase, showing its concepts, workflow, activities, artifacts and guidelines. Furthermore, it aims to meet important standards such as PMI, QAI and IEEE 829, to ensure the quality of documentation in the development phase of testing.
Thus, this paper presents the testing strategy, focusing on the process V & V (Verification and Validation), the Software Testing Techniques, covering projects of test cases
and test patterns. We applied and studied Test phases (strategies and testing techniques) in the evaluation of a real project.
Despite some restrictions by the development sector working with the proposed documentation stage, we conclude that with the adoption of strategies and techniques, we can get quality of software applications running with fewest possible errors, better operation of software maintenance and control over development within lower cost, schedule and quality levels desired.

Keywords: Strategies for Test Techniques, Test, V & V

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

BVA	Boundary Value Analysis
CMMI	Capability Maturity Model Integration
DFD	Diagrama de Fluxo de Dados
EPCT	Especificação e Procedimento de Casos de Teste
GUI	Graphical User Interface
HD	HelpDesk
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
IGU	Interface Gráfica do Usuário
LDT	Log de Teste
MTTR	Mean Time-to-Repair (Tempo médio para reparo)
OO	Orientado a Objeto
PDL	Perl Data Language
PTM	Plano de Teste Mestre
RVT's	Relatório de Visita Técnica
TMM	Test Maturity Model
UML	Unified Modeling Language
V&V	Verificação e Validação

SUMÁRIO

1 INTRODUÇÃO

1.1 Objetivos

O tema desenvolvido na pesquisa aqui apresentada é o Teste de *Software* que compõe a fase de Validação tendo como área de concentração a Engenharia de *Software*.

Objetivo geral deste trabalho é apresentar as estratégias e técnicas utilizadas bem como o processo V&V (Verificação e Validação), avaliando seu impacto na geração de *software* de qualidade.

Esta fase trata da investigação do *software*, com o objetivo de fornecer informações sobre sua qualidade em relação ao contexto em que ele deve operar.

Os objetivos específicos deste trabalho são: apresentar os principais conceitos bem como os princípios de teste, estudar suas estratégias e técnicas e avaliá-las, além de fornecer uma síntese das fases, papéis e artefatos.

Mediante a estes conceitos e conhecimentos, ser capaz de: apresentar os fundamentos dos processos de Teste, a importância do Ciclo de Testes independente, descrever suas dimensões, conhecer todas as variáveis de um Teste bem planejado e as habilidades necessárias para sua execução.

Pretende-se com esta pesquisa estudar a fase de Testes mediante a opinião e conhecimentos de alguns autores nesta área como: Pressman[1] e Sommerville [2].

O objetivo final do trabalho é ter uma visão mais abrangente da fase de testes, identificar seus pontos positivos e negativos no desenvolvimento de *software*s.

---

[1] **Roger S. Pressman** é em engenheiro de software, escritor e consultor, norte-americano, presidente da R.S. Pressman & Associates, que aborda os temas Estratégias de Teste de Software e Técnicas de Teste de Software em seu livro *Engenharia de Software*. 6ª Ed.

[2] **Ian Sommerville F.,** cientista da computação e autor britânico, aborda os temas: Teste de Software e Verificação e Validação em seu livro *Engenharia de Software*. 8° Ed.

1.2 Justificativa

A justificativa para este trabalho é a importância de conhecer as principais estratégias e técnicas de testes, quesito fundamental para garantir ao cliente a aquisição de *softwares* de qualidade.

1.3 Estrutura do Trabalho

Este trabalho propõe apresentar a parte teórica e breve relato de uma aplicação prática, utilizando os conceitos e práticas estudados neste trabalho. Para isto a estrutura do trabalho será da seguinte forma: Estratégias de Testes e Técnicas de Testes do ponto de vista de Pressman. Mediante os conceitos de Sommerville será apresentado o processo V&V (Verificação e Validação bem como os conceitos de Testes de *Software*.

A idéia é que através desse levantamento seja possível ter uma visão de como seria aplicar a fase Testes de *Software* adotando tais estratégias e técnicas na empresa em que trabalho. Serão disponibilizados ao final desta pesquisa os artefatos usados na aplicação da fase de teste como: Plano de Teste, Especificação e Procedimentos de Casos de Teste e Log de Teste. Tais artefatos seguirão o padrão IEEE 829, adaptados conforme a necessidade do projeto em questão.

Será disponibilizado tal conhecimento em nossa intranet, para que tal levantamento esteja ao acesso dos interessados com intuito de obter *software*s de qualidade.

O Plano de Teste irá apresentar a importância do planejamento para elaboração de casos de teste, irá abordar as estratégias de testes bem como aplicar alguma das técnicas discutidas no item: Técnicas de Teste no capítulo 7. O relatório de Especificação e Procedimentos de Casos de Teste contém os cenários e casos de teste a serem aplicados no projeto em questão. E por fim o Relatório de Log de Teste apresentará uma síntese do que foi obtido na realização dos casos de teste.

Através da conclusão será possível relatar alguns benefícios bem como restrições ou dificuldades na hipótese de ser usar tais estratégias e técnicas.

2 VISÃO GERAL

Aqui será apresentada uma visão geral do tema abordado nesta pesquisa: Testes de *Software*, abordando os principais temas deste trabalho: Estratégias de Testes e Técnicas de Testes.

Antes vale a pena ressaltar que o teste oferece o último reduto no qual a qualidade pode ser avaliada e erros podem ser descobertos. A qualidade é incorporada ao *software* durante ao processo de engenharia, sendo a aplicação adequada de métodos e ferramentas, revisões técnicas formais efetivas e gerência e medição sólida, todas levam a qualidade que é confirmada durante o teste.

Estratégias de Testes

De acordo com Pressman (2006, p. 288) "Uma estratégia de teste integra métodos de projeto de casos de testes em uma série bem planejada de passos que resultam na construção bem sucedida de software.", elas vêem fornecer os passos a serem conduzidos na realização dos Testes. São objetos que constituem as estratégias de testes:

Planejamento de Testes, Projeto de Casos de Testes, Execução de Teste tendo como resultado: a coleta e avaliação dos dados. A princípio as estratégias de testes são: Teste de Unidade, Teste de Integração, composto pelos testes de Regressão e Fumaça, Teste de Validação: Alfa e Beta e, por fim, Teste de Sistemas que engloba os testes de: Recuperação, Segurança, Estresse e Desempenho.

Responsáveis: Gerente de Projeto, Engenheiros de *Software* e Especialistas em Testes e finalmente os clientes (stakeholders) da aplicação.

Importância: A importância das estratégias é evitar tempo desperdiçado, esforço desnecessário, infiltração de erros sem serem descobertos.

Passos: Primeiramente os testes focalizam um único componente ou um pequeno número de componentes relacionados, aplicados para descobrir erros nos dados e na lógica de processamento. Em seguida testes de alto nível são executados para descobrir erros na satisfação dos requisitos dos clientes. E por fim os erros encontrados devem ser diagnosticados e corrigidos usando o processo de depuração.

Produtos: Os produtos são **Especificação do Teste e o Plano de Testes** – documenta a abordagem da equipe de *software* para o teste, definindo o **Plano de**

Testes – descreve uma estratégia global e um procedimento que define passos específicos de teste e os testes que serão conduzidos. Enfim as estratégia de teste são determinadas através do gerenciamento de riscos e baseadas em requisitos.

Técnicas de Testes

As técnicas de testes são diretrizes sistemáticas para projetar testes que exercitam a lógica e as interfaces de cada componente, através dos domínios de entrada e saída do programa para descobrir erro na função, no componente e no desempenho do programa.

Responsáveis: Nos primeiros estágios dos testes um engenheiro realiza todos os testes, a medida que os testes progridem, especialistas podem ser envolvidos.

Importância: O programa deve ser executado antes de chegar ao cliente, com o objetivo especifico de encontrar e remover todos os erros, testes devem ser conduzidos sistematicamente, para isto casos de testes devem ser projetados usando técnicas disciplinadas.

Passos: Nas aplicações convencionais o *software* é testado sob duas perspectivas diferentes: Caixa-Branca – a lógica interna do programa é exercitada usando técnicas de projeto de casos de teste. Caixa-Preta – requisitos de *software* são exercitados por meio de projeto de casos de teste. Obs.: para aplicações orientadas a objetos o teste começa antes da existência do código. Nos dois casos o objetivo é encontrar o maior número de erros com a menor quantidade de esforço e tempo. Casos de uso ajudam no projeto de testes para descobrir erros no âmbito da validação do software.

Produtos: Um conjunto de casos de teste é projetado para exercitar a lógica interna, interfaces, colaborações de componentes e os requisitos externos são projetados e documentados, os resultados esperados são definidos e os resultados reais são registrados.

Quais são as características da testabilidade?

Pressman cita alguns conceitos de James Bach [3]sobre a testabilidade:

A testabilidade de software é simplesmente quão fácil um programa de computador pode ser testado. Veja algumas das características que levam a um software testável:

[3] Os parágrafos citados acima são usados por Pressman com devidas permissão de James Bach, C de 1994, e foram adaptados de material que apareceu originalmente em um pôster no grupo de notícias comp.software-eng.

Operabilidade – “Quanto melhor funciona, mais eficiente pode ser testado.” Se um sistema é projetado e implementado com qualidade em mente, poucos defeitos vão bloquear a execução dos testes, permitindo que o teste progrida sem arrancos.

Observabilidade – Entradas fornecidas como parte do teste produzem saída distinta. Estado e variáveis do sistema são visíveis ou consultáveis durante a execução. Saída incorreta é facilmente identificada. Erros internos são automaticamente detectados. O código –fonte é acessível.

Controlabilidade – “Quanto melhor você pode controlar o software, mas o teste pode ser automatizado e otimizado.” Estados e variáveis do *software* e do *hardware* podem ser controlados diretamente pelo engenheiro de teste. Testes podem ser especificados, automatizados e reproduzidos convenientemente.

Decomponibilidade – “Controlando o escopo do teste, podemos isolar problemas mais rapidamente e realizar retestagem mais racionalmente.” O sistema de software é construído por meio de módulos independentes, que podem ser testados independente.

Simplicidade – “Quanto menos houver a testar, mais rapidamente podemos testá-los.” O programa deve exibir simplicidade funcional (por exemplo, o conjunto de características é o mínimo necessário para satisfazer aos requisitos), simplicidade estrutural (por exemplo, a arquitetura é modularizada para limitar a propagação de defeitos), simplicidade do código (por exemplo, uma norma de codificação é adotada para facilitar a inspeção e manutenção).

Estabilidade - “Quanto menos modificações, menos interrupções no teste.” Modificações no *software* não são freqüentes, controladas quando ocorrem e não invalidam os testes existentes. O *software* recupera-se bem de falhas.

Compreensibilidade – “Quanto mais informações temos, mais racionalmente vamos testar.” O projeto arquitetural e as dependências entre componentes internos, externos e compartilhados são bem compreendidos. Documentação técnica é acessível instantaneamente, bem organizada, especifica, detalhada e precisa. Modificações ao projeto são comunicadas aos testadores.

Os atributos citados acima podem ser usados por um engenheiro de *software* para desenvolver uma configuração de *software* (isto é, programas, dados e documentos) que é fácil de testar.

Vimos aqui um panorama do que vem a ser estratégias e técnicas de testes e as características de testabilidade. A seguir serão apresentados os conceitos chaves da pesquisa.

3 CONCEITOS

Nesta seção serão abordados os principais conceitos referentes a Testes de *Software*.

Teste – processo de executar um programa com a finalidade de encontrar erros. [MYERS, 1979].

Segundo [PRESSMAN, 2006] é o conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente.

Teste de baixo nível – verifica se um pequeno segmento de código-fonte foi corretamente implementado.

Teste de alto nível – valida as principais funções do sistema com base nos requisitos do cliente.

Verificação – conjunto de atividades que garante que o *software* implementa corretamente a função específica.

Validação – conjunto de atividades que garante que o *software* construído corresponde aos requisitos do cliente.

Depuração – processo de corrigir os erros encontrados na execução dos testes.

Estratégias de teste – são métodos de projeto de casos de testes em uma série bem planejada de passos, elas vêem fornecer os passos a serem conduzidos na realização dos Testes.

Técnicas de Teste - diretrizes sistemáticas para projetar testes que exercitam a lógica e as interfaces de cada componente, através dos domínios de entrada e saída do programa para descobrir erro na função, no componente e no desempenho do programa. Com o objetivo de encontrar erros, sendo um bom teste aquele que tem alta probabilidade de encontrar um erro.

Pseudocontrolador – programa principal que aceita dados do caso de teste passa tais dados ao componente a ser testado e imprime os resultados relevantes.

Pseudocontrolado (pseudo subprograma) – servem para substituir módulos que são subordinados (chamado pelo) ao componente a ser testado. Este usa a interface dos módulos subordinados, faz no mínimo manipulação de dados, fornece verificação da entrada e devolve o controle ao módulo que está sendo testado.

Foram apresentados aqui os principais conceitos abordados no desenvolvimento da pesquisa. No tópico seguinte serão abordadas as estratégias de testes.

4 ESTRATÉGIAS DE TESTES

Serão apresentadas aqui as principais estratégias de teste de *software*.

É importante ressaltarmos que a escolha do tipo de teste vai depender das características do *software* e do cronograma do projeto. Uma abordagem combinada (teste sanduiche), que usa testes descendentes para os níveis mais altos da estrutura do programa acoplada com testes ascendentes para os níveis subordinados, talvez seja a melhor opção.

4.1 Teste de Unidade

Este tipo de testes focaliza a menor unidade de projeto do *software* – componente ou módulo. Ele enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente. Algumas considerações a respeito do teste de unidade:

A interface é testada para garantir que a informação flui adequadamente para dentro e para fora da unidade de programa que esta sendo testada.

A estrutura de dados é examinada para garantir que os dados armazenados temporariamente mantém sua integridade durante todos os passos de execução de um algoritmo, sendo todos os caminhos independentes (caminhos básicos) executados pelo menos uma vez. As condições-limite são testadas para garantir que o módulo opere adequadamente nos limiares estabelecidos para limitar ou restringir o processamento. E por fim todos os caminhos de manipulação de erros são testados.

O teste de unidade será simplificado quando um componente com alta coesão é projetado. Quando apenas uma função é implementada por um componente, o número de casos de teste é reduzido e os erros podem ser mais facilmente previstos e descobertos. Este tipo de teste utiliza técnicas de Caixa-Branca.

4.1.1 Teste Seletivo de caminhos de execução

Os casos de teste devem ser criados para descobrir erros devidos a cálculos errados, comparações incorretas ou fluxo de controle inadequados. Veja abaixo erros comuns no cálculo:

1) Precedência aritmética mal entendida ou incorreta;
2) Operações em modo misto;
3) Inicialização incorreta;
4) Falta de precisão e
5) Representação incorreta de uma expressão simbólica.

Em se tratando de comparação e fluxo de controle podemos citar:

1) Comparação de tipos de dados diferentes;
2) Operadores ou precedência lógica incorretos;
3) Expectativa de igualdade quando o erro de precisão torna a igualdade improvável;
4) Comparação incorreta de variáveis;
5) Terminação de ciclo inadequada ou inexistente;
6) Falha na saída quando iteração divergente é encontrada e
7) Variáveis de ciclo inadequadamente modificadas.

4.1.2 Teste nos limites

O *software* falha freqüentemente nos seus limites. Os erros ocorrem quando o *n-ésimo* elemento de um vetor de dimensão *n* é processado, quando a *i-ésima* repetição de um ciclo com *i* passagem é chamado, quando o valor máximo e mínimo é permitido é encontrado. Aqui casos de teste que exercitam estrutura de dados, fluxo de controle e valores de dados imediatamente abaixo, imediatamente acima e no máximo e no mínimo provavelmente descobrirão erro.

É necessário analisar entre os erros potenciais que devem ser testados quando a manipulação de erros é avaliada temos:

1) A descrição de erro é ininteligível;
2) O erro mencionado não corresponde ao erro encontrado;

3) A condição de erro provoca a intervenção do sistema antes da manipulação do erro;
4) O processamento da condição de exceção está incorreto;
5) A descrição do erro não fornece informação suficiente para ajudar na localização da causa do erro.

4.1.3 Procedimentos de Teste de Unidade

Este tipo de teste pode ser realizado antes que o código seja iniciado (uma abordagem ágil preferida) ou depois do código-fonte ter sido desenvolvido.
A revisão de informação de projeto fornece diretrizes para a criação dos casos de teste que devem ser acoplado a um conjunto de resultado esperado.
Considerando que um componente não é um programa isolado, o *software* para um pseudocontrolador (driver) e/ou pseudocontrolado (stub) deve ser desenvolvido para cada teste de unidade.

Aqui um pseudocontrolador (programa principal) que aceita dados do caso de teste passa tais dados ao componente a ser testado e imprime os resultados relevantes.

Enquanto que o pseudocontrolado (pseudo subprograma) substitui os módulos que são subordinados (chamado pelo) ao componente a ser testado. Este usa a interface dos módulos subordinados, faz no mínimo manipulação de dados, fornece verificação da entrada e devolve o controle ao módulo que está sendo testado.

Estes precisam ser escritos, mas não são entregues ao produto final do software, são mantidos de forma bem simples, as despesas indiretas reais são relativamente baixas, mesmo assim muitos componentes não podem ser testados no nível de unidade do modo adequado com o software adicional “simples”, podendo ser adiado e feito junto ao teste de integração.

Por fim **Teste de unidade** é toda a aplicação de teste nas assinaturas de entradas e saídas de um sistema que consiste em validar dados válidos e inválidos via Input/Output (entrada/saída), ou seja, todas as entradas e saídas existentes na programação. Considerando assim avaliar as entradas e saídas nos seguintes aspectos:

Válidos (dados comuns ao sistema), Inválidos (dados não comuns ao sistema), domínio (só permite dados com formatação igual a que será armazenada, exemplo Telefone), tipos de valores (exemplo cálculo do dia da páscoa), data (deve ser feita conforme o padrão da região de uso), número (devem ser tratados no formato correto e

com as regras para a região especificadas pelo usuário. Exemplo: validar valores negativos ou sinais), senha (deve atender as normas da empresa. Exemplo: não deve ser exibida durante a digitação somente com caractere representativo), campos especiais (como CPF, CNPJ, CEP entre outros). Além disso, é preciso ficar atento aos valores para teste como: sintaxes da linguagem (muitas linguagem podem ter problemas ao tratar valores do tipo objeto, string e outros. Tais linguagem tem tratamentos especiais para estes valores para assim evitar erros ou em alguns casos não tem tratamento, pois o erro é na codificação), erro na linguagem (um exemplo claro de erro na linguagem são sistemas feitos na própria base de dados que não permitem pesquisas usando termos como *else*, *while*, *for* por serem palavras reservadas da linguagem, atualmente a maior parte dos sistema não apresenta erros com termos reservados), alteração de valores (algumas linguagens ao receberem determinados valores alteram seus dados internos com tratamentos automáticos no geral elas possuem travas para estes tratamentos. Exemplo de valores 0x00, 00FF, 1.1, 1,1, 1^2 e etc), interpretação de valores (algumas linguagens interpretam diretamente valores sem tratar passando eles para outra aplicação em uso), cálculos mesclados (geralmente gerados pela entrada de dados inválidos pelo usuário, isto deve ser tratado), formatação de região ou não, medidas, datas, navegação (pode ser tanto a navegação sobre o sistema ou a navegação entre sistemas. Sendo que somente a navegação dentro do bloco validada como teste unitário. Os itens internos ao sistema e entre sistema são testes de integração) e por fim local (navegação entre os campos ou funcionalidades da tela, esta deve sempre atender a ordem esquerda para direita de cima para baixo. Caso o usuário use outro padrão ele deve ter especificado previamente pois esta fora dos padrões de mercado. A navegação local deve também validar os valores entre campos e seus resultados. Atentar para alteração de valores em campos diferentes do usado, algumas regras pode força as alterações de campos secundários devem validar todas as interações que geram estas alterações com dados validos e inválidos).

4.1.3.1 Sugestão para teste unitário

Aspecto geral: selecionar módulos críticos e aqueles que têm complexidade ciclomática alta fazendo o teste de unidade apenas neles. Além dessa dica existem seis regras básicas para testes unitários:

1º Revisão de código.

2º Existe erro sem tratamento ou log para seu registro.

3º Existe erro na tela do usuário.

4º O usuário esta recebendo somente mensagens claras e sem vínculos técnicos conforme os padrões da empresa.

5º Todo processo lento tem apresentação de status de progresso conforme os padrões da empresa.

6º Os padrões de formatação estão atendendo a região que o usuário especificou: BR,EUA e etc.

4.1.4 Teste de Unidade no Contexto OO

Pressman também aborda este tipo de teste no contexto OO, pois em se tratando de *softwares* o conceito de unidade se modifica. O encapsulamento guia a definição de classes e objetos. Cada classe e cada instância de uma classe (objeto) empacotam os atributos (dados) e as operações (funções) que manipulam estes dados. Sendo assim uma classe encapsula o foco do teste de unidade. Neste contexto as operações são a menos unidade testável, considerando que tais operações podem ser diferentes uma das outras, dessa forma não podemos testar uma operação isoladamente, visão convencional do teste de unidade, e sim como parte de uma classe. Enfim este tipo de teste é guiado pelas operações encapsuladas na classe e pelo seu estado de comportamento.

4.2 Teste de Integração

É uma técnica sistemática para construir a arquitetura do *software* enquanto ao mesmo tempo, conduz testes para descobrir erros associados às interfaces, tendo como objetivo a partir de componentes testados no nível de unidade, construir uma estrutura de programa determinada pelo projeto.

Aqui o programa inteiro é testado de uma só vez (abordagem *big-bang*) podendo dificultar a correção, pois o isolamento dos erros fica difícil pelo vasto espaço do

programa inteiro. A integração incremental é a conduta de contrária citada anteriormente, onde o programa é construído e testado em pequenos incrementos, sendo os erros mais fáceis de isolar e corrigir.

Neste método temos duas estratégias de testes:

Descendente (*top-down*) – Os módulos são integrados movendo-se descendentemente pela hierarquia de controle, começando com o módulo de controle principal (programa principal). Os módulos subordinados (e os que são subordinados em última instância) ao módulo de controle principal são incorporados à estrutura de maneira primeiro-em-profundidade ou primeiro-em-largura.

A integração primeiro-em-profundidade integra todos os componentes, em um caminho de controle principal da estrutura.

A integração primeiro-em-largura incorpora todos os componentes diretamente subordinados em cada nível, movendo-se ao longo da estrutura horizontalmente.

Passos para integração descendente:

1) Módulo de Controle principal é usado como pseudocontrolador do teste, e pseudocontrolados são substituídos por todos os componentes diretamente subordinados ao módulo de controle principal;
2) Dependendo da abordagem escolhida em profundidade ou em largura, os pseudocontrolados são substituídos, um de cada vez pelos componentes reais;
3) Testes são conduzidos a medida que cada componente é integrado;
4) Ao término de cada conjunto de testes, outros pseudocontrolado é substituído pelo componente real;
5) O teste de regressão pode ser conduzido para garantir que novos erros não tenham sido introduzidos.

Esta estratégia verifica os principais pontos de controle ou decisão logo no início do processo de teste. Em uma estrutura de programa bem fatorada, a tomada de decisão ocorre nos altos níveis da hierarquia e assim é encontrada primeiro.

Fica claro que a demonstração da capacidade funcional, logo no inicio, é uma fator de confiança para o desenvolvedor e para o cliente.

Pode surgir problemas logísticos nesta etapa, isto ocorre quando o processamento em níveis baixos são necessários para testar adequadamente níveis

superiores, onde nenhum dado significativo flui para cima na estrutura do programa. Dessa forma os testadores têm três escolhas:

1) Adiar muitos testes até que pseudocontrolados sejam substituídos pelos módulos reais, aqui perdemos algum controle sobre a correspondência entre testes específicos e a incorporação de módulos específicos, dificultando determinar a causa de erros violando a abordagem descendente;
2) Desenvolver pseudocontrolados que realizam funções limitadas, simulando o módulo real, esta pode levar a despesas indiretas significativas, na medida em que pseudocontrolados tornam se mais e mais complexos;
3) Integrar o *software* de baixo para cima, na hierarquia, esta conhecida como abordagem ascendente (*botton-up*) será nossa próxima discussão.

Ascendente (*botton-up*) - inicia-se a construção e testes de módulos atômicos (componentes nos níveis mais baixos). Como os componentes são integrados de baixo para cima, o processamento necessário para os componentes subordinados em um determinado nível está sempre disponível e as necessidades de pseudocontrolados é eliminada.

Veja os passos para esta abordagem:

1) Componentes de baixo nível são combinados em agregados (*clusters* – construções), realizando uma subfunção especifica do *software*.
2) Um pseudocontrolado (*driver*, programa de controle para teste), é desenvolvido para dar entrada e saída do caso de teste;
3) O agregado é testado;
4) Pseudocontroladores são removidos e agregados são combinados movendo-se para cima na estrutura do programa.

A medida que a integração se move para cima, o número de peseucocontroladores de teste diminui.

Dentro do teste de integração temos mais duas abordagens de teste:

4.2.1 Teste de regressão

É a re-execução de algum subconjunto de testes que já foi conduzido para que as modificações não propaguem efeitos colaterais indesejáveis. É a atividade que garante que modificações (devidas ao teste ou a outras razões) não introduzam comportamento indesejável ou erros adicionais.

Este tipo de teste pode ser conduzido manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automatizadas de captação/reexecução, tais ferramentas permitem o engenheiro de software captar casos de teste e resultados para subseqüente reexecução e comparação.

A suíte de teste de regressão (subconjunto de testes a ser executado) contém 3 classes de casos de teste:

Uma amostra representativa de testes que vão exercitar todas as funções do *software*;

Testes adicionais que focalizam funções do *software*, que serão provavelmente afetadas pela modificação;

Testes que focalizam os componentes de *software* que foram modificados.

É comum que a medida que o teste de integração prossegue, o número de testes de regressão pode crescer significativamente.

Esta deve ser projetada para incluir apenas aqueles testes que cuidam de uma ou mais classes de erros em cada uma das principais funções do programa. Não é prático e eficiente re-executar cada teste para cada função do programa, toda vez que ocorre uma modificação.

Tem como propósito garantir que os defeitos encontrados foram corrigidos e que as correções ou inserções de novos códigos em determinados locais do software não afetaram outras partes inalteradas do produto. Trata de re-testar o teste. É necessário ter ferramentas para execução do teste de regreção, isto porque é inviável testar novamente todo o Software.

4.2.2 Teste fumaça

É uma estratégia usada quando produtos de *software* estão sendo desenvolvidos. Projetado como mecanismo de marca-passo para projetos de prazo crítico, permitindo que a equipe de *software* avalie seu projeto em bases freqüentes. Esta estratégia abrange as seguintes atividades:

Os componentes de *software* que foram traduzidos para código são integrados em uma "construção" que inclui todos os arquivos de dados, bibliotecas, módulos reusáveis e componentes submetidos a engenharia, que são necessários para implementar uma ou mais funções do produto.

Uma série de erros é projetada para expor erros que impeçam a construção de desempenhar adequadamente a sua função. O objetivo é descobrir erros "bloqueadores" (*show stopper*) que têm a maior probabilidade de colocar o projeto fora do cronograma.

A construção é integrada em outras construções e o produto inteiro (na sua forma atual) passa pelo testes fumaça diariamente, sendo que a abordagem de integração pode ser descendente ou ascendente.

Testes freqüentes dão, tanto aos gerentes quanto aos profissionais, uma avaliação realística do progresso do teste de integração.

Em suma é considerado uma estratégia de integração constante, onde *software* é reconstruído (com adição de novos componentes) e submetido a teste todos os dias. Este tipo de teste deve exercitar o sistema de ponta a ponta, sem precisar ser exaustivo, mas deve ser capaz de expor os problemas principais, no entanto deve ser rigoroso.

Geralmente aplicado a projetos de engenharia complexos e de prazos críticos, uns dos benefícios na atuação deste teste são:

Risco de integração minimizado – Por este tipo de teste ser conduzido diariamente erros são descobertos no início, reduzindo o impacto no cronograma quando erros são descobertos.

Qualidade do produto final é aperfeiçoada – é provável que tal erro descubra tanto erros funcionais quanto defeitos de projeto arquitetural e no nível de componente. Corrigindo tais erros no início o resultado será um produto de melhor qualidade.

Diagnósticos e correção de erros são simplificados – o teste fumaça é associado com novos incrementos no software, concluindo-se então que tais incrementos são possíveis de causar erros.

Progresso é fácil de avaliar - isto porque quanto mais o *software* tenha sido integrado e demonstrado que é funcional dando assim uma boa indicação que está ocorrendo progresso.

Veja as vantagens e desvantagens relativas do teste de integração descendente versus ascendentes:

Teste de Integração	Vantagens	Desvantagens
Ascendente	O Projeto de Casos de Teste é facilitado e pela ausência de pseudocontrolados.	"O programa como uma entidade, não existe até que o último módulo seja adicionado" [MYERS, 1979]
Descendente	Vantagem de testar logo as principais funções de controle.	Necessidades de pseudocontrolados ocasionando dificuldades nos testes.

Tabela 1 Vantagens e desvantagens relativas do teste de integração descendente versus ascendentes – Retirado do livro *Engenharia de Software,* cap. 13 - Pressman.

A medida que o teste de integração é conduzido, o testador deve identificar os módulos críticos que possui uma ou mais das seguintes características: abordar vários requisitos do *software*, tem um alto nível de controle (situa-se em um ponto relativamente alto na estrutura do programa); é complexo ou propenso a erro ou tem requisitos de desempenho bem definidos. Este tipo de módulos deve ser testado tão cedo quanto possível.

4.2.3 Documentação

Especificação de Teste – documenta um plano global para integração do *software* e uma descrição dos testes específicos, composto do: Plano de Teste e Procedimentos de Teste.

Plano de Teste – descreve a estratégia global de integração. O teste é dividido em fases e construções que tratam de características funcionais e comportamentos específicos do *software*. Critérios e testes correspondentes são aplicados a todas as fases as quais são:

Integridade da interface – interfaces internas e externas são testadas a medida que cada módulo é agregado à estrutura;

Validade Funcional – testes são projetados para descobrir erros funcionais;

Conteúdo Informacional – testes são projetados para descobrir erros associados a estrutura de dados locais ou globais e

Desempenho – testes verificam limites de desempenho estabelecidos durante o projeto.

O cronograma, o desenvolvimento de *software* de uso geral e tópicos relacionados fazem parte do Plano de Teste. Datas iniciais e finais são estabelecidas para cada fase e "janelas de disponibilidade" para módulos são submetidos a teste de unidade. Uma breve descrição do *software* de uso geral (pseudocontroladores e pseudocontrolados), o ambiente e recursos de testes também são descritos. Configuração de hardware não usual, simuladores exóticos e ferramentas ou técnicas de teste especiais são alguns de muitos tópicos que podem ser discutidos.

Procedimentos de Teste – este é descrito de forma detalhada após o Plano de Teste. A ordem de integração e os testes correspondentes em cada passo de integração são descritos. Inclui também uma lista de todos os testes bem como os resultados esperados.

Relatórios de Testes – anexado a Especificação de Teste se desejado, podendo ser vital durante a manutenção do *software.* Este item será detalhado no tópico dez desta pesquisa.

4.2.4 Teste de Integração no Contexto OO

Aqui este tipo de teste tem pouco significado. A integração de operações, uma de cada vez em uma classe (abordagem incremental) é impossível por causa das interações diretas e indiretas dos componentes de classes. Temos assim duas técnicas:

Thread-based testing – Teste baseado no caminho de execução, onde integra o conjunto de classes necessárias para responder uma entrada ou evento do sistema. Onde cada

caminho é testado individualmente. O teste de regressão é aplicado para garantir que nenhum efeito colateral ocorra.

Use-Based Testing - Teste baseado no uso. Começa a construção do sistema testando as classes chamadas de classes independentes que usa muito pouco ou quase nenhumas classes servidoras. Em seguidas a camada de classes dependentes são testadas, assim por diante até que todo sistema seja construído.

Aqui pseudocontroladores são usados para testar operações nos níveis mais baixos e grupos inteiros de classes, podendo ser usado para substituir interface com o usuário de modo que testes da funcionalidade do sistema possam ser conduzidos antes da implementação da interface. Os pseudocontrolados são usados em situações em que colaborações entre classes é necessária, mas uma ou mais classes colaboradoras não foram totalmente implementada.

O teste agregado é uma etapa para o teste de Integração OO. Um agregado de classes colaboradoras (exame dos modelos CRC e objeto relacional) é exercitado projetando-se casos de teste para descobrir erros nas colaborações.

Uma estratégia importante aqui é baseada no caminho de execução que são conjuntos de classes que respondem a uma entrada ou evento.

4.3 Teste de Validação – Teste de Aceitação

Este tipo de teste começa no fim do teste de Integração, aqui o *software* está completamente montado e os erros de interface já foram descobertos e corrigidos, focaliza ações visíveis ao usuário e saídas do sistema reconhecidas pelo usuário, não existindo distinção entre o software convencional e o orientado a objetos.

Validação - o *software* funciona de um modo que pode ser razoavelmente esperado pelo cliente. O teste de validação é feito mediante a Especificação dos Requisitos de *Software* – documento que descreve todos os atributos do *software* visíveis aos usuários, este documento contém uma seção chamada *Critério de Validação*, base para a abordagem do respectivo teste.

A validação do *software* é conseguida por intermédio de uma série de testes que demonstram conformidade com os requisitos.

4.3.1 Documentação

Plano de Teste – delineia as classes de teste a ser conduzidas.

Procedimento de Teste – define os casos de teste específicos.

Ambos são projetados para garantir que todos os requisitos funcionais sejam satisfeitos, bem como as características comportamentais sejam conseguidas, todos os requisitos de desempenho sejam alcançados, a documentação esteja correta, usabilidade e outros requisitos sejam satisfeitos.

Após cada caso de teste tenha sido executado, uma de duas possíveis condições se realiza:

A característica de função ou desempenho satisfaz a especificação e é aceita ou é descoberto um desvio da especificação e uma lista de deficiência é criada, estes raramente podem ser corrigidos antes da entrega programada.

A Revisão da Configuração, conhecida também como *auditoria* é um importante elemento do processo de validação, esta garante que todos os elementos da configuração de *software* tenha sido adequadamente desenvolvidos, estejam catalogados e tenham os detalhes necessários para apoiar a fase de suporte de ciclo de vida do *software*.

É impossível para o desenvolvedor de *software* prever como o cliente usará realmente um programa, o que parece claro para o testador pode ser inteligível para um usuário no campo. Quando um *software* é feito sob encomenda, uma serie de testes de aceitação é conduzido para permitir ao cliente validar todos os requisitos, este conduzido pelo usuário final, podendo acontecer ao longo de um período de semanas ou meses, descobrindo conseqüentemente erros cumulativos que poderiam degradar o sistema ao longo do tempo. Se o *software* é desenvolvido como um produto a ser usado por vários clientes, não é prático realizar testes formais de aceitação com cada um. Para isto utilizam-se os seguintes testes: alfa e beta.

Testes Alfa – conduzido na instalação do desenvolvedor com os usuários finais. Sendo o *software* usado em um ambiente natural com o desenvolvedor acompanhando os usuários e registrando erros e problemas de uso, conduzido em um ambiente controlado.

Teste Beta – conduzido nas instalações dos usuários finais, sem a presença do desenvolvedor. O cliente registra todos os problemas reais ou imaginários e os relata ao desenvolvedor responsável em fazer as devidas correções e publicar uma nova versão.
Podemos tratar aqui três dimensões:
Funcionalidade – quais as regras de negócio devem contemplar o Software.
Performance – envolve tempo mínimo de resposta das funções.
Qualidade - envolve atributos de confiança do Software.
Neste contexto aproveitaremos para falar do teste de funcionalidade.

4.3.2 Teste de funcionalidade

O teste funcional tem por meta verificar se o *software* executa corretamente suas funções, se a implementação das regras de negócio foi apropriada e se o sistema é capaz de sustentar sua correta execução por um período contínuo de uso.
Baseado nas técnicas de caixa-preta. Analisa as saídas e resultados. Os testes funcionais são baseados nos procedimentos determinados nos casos de teste.
Tem como objetivos: **a**ssegurar a funcionalidade do sistema, incluindo entrada de dados, processamento e resposta, verificar se os requisitos dos usuários estão implementados e se atendem os usuários, verificar se o sistema funciona corretamente após um período contínuo de utilização.
Deve ser usado em qualquer sistema devendo ter suas funcionalidades testadas, pode ser usado desde a fase de especificação de requisitos até a fase de operação do sistema.

4.4 Teste de Sistema

A seguir iremos abordar o teste de sistema.

Este tipo de teste é definido como uma série de diferentes testes cuja finalidade principal é exercitar por completo o sistema baseado em computador.

É importante falarmos que o *software* é apenas um elemento de um sistema maior, ele é incorporado a outros elementos do sistema (*hardware*, pessoal e informação) e uma série de teste de integração e validação do sistema é conduzida. Este tipo de teste sai do escopo do processo de *software*, passos tomados durante o projeto e teste de *software* podem melhorar muito a probabilidade de integração de *software* bem

sucedida no sistema maior. É recomendado que o engenheiro de *software* antecipe problemas potenciais de interface e projete caminhos de manipulação de erro que testem toda informação que chega de outros elementos do sistema, conduzir uma série de testes que simulam maus dados ou outros erros em potencial na interface do *software*, registrar os resultados dos testes para serem usados, participar do planejamento e projeto de testes de sistemas para garantir que o *software* seja adequadamente testado.

Teste de Sistemas é uma série de testes, com finalidade principal exercitar por completo o sistema baseado em computador, focalizando o funcionamento do sistema como um todo. Executado mediante a especificação do sistema, com intuito de verificar se as funcionalidades estão sendo entregues de acordo com que o cliente solicitou.
Lan considera este tipo de teste como a integração de dois ou mais componentes que implementam funções ou características do sistema integrado. No processo de desenvolvimento iterativo, este tipo de teste concentra-se no teste de um incremento que será entregue ao cliente, já em cascata o teste concentra-se no teste de todo o sistema.

Este tipo de teste é realizado dentro de um ambiente controlado (servidor, equipe, máquinas, clientes, *software*, *browser*) Nas seções seguintes serão apresentados os tipos de teste de sistema:

4.4.1 Teste de Recuperação – Teste de Contingência

Muitos sistemas baseados em computador devem-se recuperar de falhas e retomar o processamento dentro de um tempo pré-especificado. Em alguns casos, um sistema deve ser tolerante a falhas, ou seja, falhas de processamento não devem causar a interrupção da função global do sistema. Em outros casos tais falhas devem ser corrigidas em um tempo especificado ou poderá ocorrer um sério prejuízo econômico.

Assim o teste de recuperação também conhecido como Testes de Contigência força o sistema a falhar de diversos modos e verifica se a recuperação é adequadamente realizada. Tanto a recuperação automática quanto a humana, o tempo médio para reparo (*mean-time-to-repair*, MTTR) é avaliado para determinar se está dentro de limites aceitáveis.

Os Objetivos são: manter o *backup* dos dados, armazenar o backup em local seguro, documentar os procedimentos de recuperação e deixar claro as responsabilidades das pessoas em caso de um desastre.

Deve ser usado sempre que a continuidade dos serviços for essencial para o negócio, as perdas devem ser estimadas sendo que a quantidade de perda potencial estipula a quantidade de esforço que irá ser empregada.

4.4.2 Teste de Segurança

Todo sistema baseado em computador que administra informações sensíveis ou causa ações que podem inadequadamente prejudicar ou beneficiar é um alvo para invasões imprópria ou ilegal, que pode ocorre por meio de *hackers,* empregados descontentes, indivíduos desonestos entre outros que tentam se dar bem mediante fatos ilícitos.

Assim o teste de segurança verifica se os mecanismos de proteção incorporados a um sistema vão de fato protegê-lo de invasão imprópria.

Durante este teste o testador desempenha os papéis de indivíduos que desejam invadir o sistema, sendo umas das tentativas: obtenção de senhas de acesso ao sistema uso de *software* projetado sob medida para destruir quais defesas construídas, causar erro no sistema de propósito, percorrer dados inseguros com intuito de descobrir a chave para entrar no sistema.

O bom teste de segurança é aquele que consegue invadir o sistema, sendo o papel do projetista tornar o custo da invasão maior do que o valor da informação que será obtida.

Tem que garantir que as funções e dados de um sistema sejam acessados apenas por atores autorizados

4.4.3 Teste de Estresse

São testes projetados para submeter programas a situações anormais, este executa um sistema de tal forma que demanda recursos em quantidade, freqüência ou volume anormais, como: testes são projetados para gerar dez interrupções por segundo, sendo a média de uma ou duas, a velocidade de entrada de dados pode ser aumentada de uma ordem de grandeza para determinar como as funções de entrada vão reagir, casos

de teste que exigem um máximo de memória são executados, casos de teste que podem causar problemas de gestão de memória são projetados, casos de teste que podem causar busca excessiva de dados residentes em disco são criado.

Este teste tem como variante o Teste de Sensibilidade, este tenta descobrir combinações de dados dentro das classes de entrada válidas, que podem causar instabilidade ou processamento inadequado. E por último temos o Teste de Desempenho.

Podemos dizer que este tipo de teste envolve a execução do sistema com baixos recursos de *hardware* e *software* ou a concorrência por estes recursos (banco de dados, rede, etc). Os objetivos são: determinar a que condições-limite de recursos o software é capaz de ser executado, verificar se o sistema é capaz de garantir tempos adequados de resposta sendo executado em condições-limite; verificar se há restrições quanto ao ambiente em que o software vai operar determinar que volumes de transação, normais e acima dos normais, podem ser processados num período de tempo esperado. Devendo assim ser usado quando não se sabe quais as condições mínimas de recursos para operacionalização do sistema, sem que haja perdas significativas.

4.4.4 Teste de desempenho – Teste de Performance

Em se tratando de sistemas de tempo real e embutidos, se o mesmo fornece a função requisitada, mas não satisfaz aos requisitos de desempenho, isto é inaceitável. Este tipo de teste é executado para testar o desempenho do *software*, durante a execução, em se tratando de um sistema integrado. O mesmo ocorre ao longo de todos os passos do processo de teste. Mesmo ao nível de unidade, o desempenho de um módulo individual pode ser avaliado á medida que testes são conduzidos. Porém o verdadeiro desempenho de um sistema não pode ser avaliado antes que todos os elementos do sistema estejam plenamente integrados. Geralmente este tipo de teste está acoplado aos testes de estresse e usualmente requerem instrumentação, tanto de *hardware* quanto de *software*. Tal instrumentação pode monitorar intervalos de execução, registrar eventos à medida que eles ocorrem e retirar amostras do estado da máquina. Dessa forma o testador pode descobrir situações que levam a degradação e possível falha do sistema.

Enfim, mede e avalia o tempo de resposta, o número de transações e outros requisitos sensíveis ao tempo de resposta do sistema.

Os objetivos são: determinar o tempo de resposta das transações, verificar se os critérios de desempenho estabelecidos estão sendo atendidos, identificar pontos de gargalo no sistema e verificar o nível de utilização do *hardware* e *software*.

Deve ser usado quando se deseja avaliar o tempo de resposta de uma funcionalidade do sistema ou de todo o sistema e quando ainda há tempo hábil de serem realizadas correções.

Enquanto Pressman aborda os testes citados acima sobre teste de sistemas, Lan aponta os seguintes testes:

Teste de Integração – os testes devem ter acesso ao código fonte, encontrando a origem do problema e identificar os componentes que devem ser depurados. Este teste verifica se os componentes funcionam realmente em conjunto, se são chamados corretamente e se transferem dados corretos no tempo correto por meio de suas interfaces.

Estratégias de teste de integração:

Top Down – consiste em identificar clusters de componentes que realizam alguma funcionalidade do sistema, e a integração desses componentes pela adição de códigos que fazem com que eles trabalhem em conjunto. Um esqueleto geral do sistema é desenvolvido primeiro, e os componentes são adicionados a ele.

Bottom Up – integrar componentes de infra-estrutura que fornecem serviços comuns, como acesso a rede e ao banco de dados e em seguida adicionar os componentes funcionais.

Testes de Regressão – quando um novo incremento é integrado, é importante executar os testes de incrementos anteriores novamente, novos testes são necessários para verificar a nova funcionalidade do sistema, ou seja, executar um conjunto de testes existentes novamente. Caso seja revelado problemas, deve-se verificar se são problemas no incremento anterior que o novo incremento revelou ou se os problemas devem-se ao incremento de funcionalidade adicionado. Este tipo de teste é impraticável sem algum apoio automatizado. Quando é utilizado um *framework* automatizado de teste como JUnit, a nova execução dos testes pode ser automatizada.

Testes de realeses (teste funcional) – uma versão do sistema que poderia ser liberada aos usuários para ser testada, para validar se o sistema atende aos requisitos e assegurar que o sistema é confiável. Este teste é composto pelo teste ‘caixa-preta’, demonstra se o sistema funciona corretamente ou não, os problemas são reportados a equipe de

desenvolvimento cujo trabalho é depurar o programa. Os clientes são envolvidos neste teste conhecido como testes de aceitação. Se o realese for bom o suficiente o cliente poderá aceitá-lo para seu uso.

Aqui deve ser apresentado a funcionalidade, o desempenho e a confiabilidade e que não apresenta falhas durante o uso normal. O sistema é tratado como uma caixa-preta, cujo comportamento pode ser somente determinado por meio do estudo de suas entradas e as saídas relacionadas.

A melhor abordagem aqui a ser usada é o teste baseado em cenários no qual é criado um número de cenários, e casos de teste são criados com base nesses cenários, para cada teste deve projetar um conjunto de entradas válidas e inválidas e que gere saídas válidas e inválidas. Aqui os cenários mais prováveis devem ser testados primeiro e os cenários incomuns ou excepcionais sejam considerados mais tarde, dedicando as partes mais usadas no sistema. Os casos de uso e diagramas de seqüências podem ser usados durante os testes de integração e de realeses.

Veja as outras estratégias de teste do ponto de vista de Lan (Ref..):

Teste de desempenho – projetados para assegurar que o sistema pode operar na carga necessária. Para testar se os requisitos de desempenho estão sendo atingidos é necessário criar um perfil operacional que é o conjunto de testes que reflete uma combinação real de trabalho a que o sistema será submetido. Experiências mostram que uma maneira eficiente de descobrir defeitos é projetar testes com base nos limites do sistema, isto significa estressar o sistema. O teste de estresse tem duas funções:

Testar o desempenho de falha do sistema. Por meio de combinações de eventos em que a carga imposta exceda a carga máxima projetada. Aqui é importante que a falha do sistema não cause o corrompimento de dados ou perda inesperadas de serviços dos usuários.

Estressar o sistema e causar o surgimento de defeitos que não seriam normalmente descobertos.

Este tipo de teste é relevante para sistemas distribuídos baseados em uma rede de processadores, que exibem degradações graves ao serem submetidos a pesadas cargas de demanda.

Teste de componentes (teste de unidades) – teste de componentes individuais do sistema. Componentes que podem ser testados neste estágio:

1) Funções ou métodos individuais de um objeto (chamadas dessas rotinas com diferentes parâmetros de entradas);
2) Classes de objeto com vários atributos e métodos;
3) Componentes compostos que constituem diferentes objetos ou funções. Esses componentes compostos têm uma interface definida usada para acessar sua funcionalidade.

Os projetos de casos de teste podem ser usados para projetar testes de funções ou métodos. A partir de seus atributos os métodos devem ser testados. Deve-se testar todas as seqüências de transição de estado.

O uso de herança dificulta projetar testes de classes de objetos. O motivo é que a operação herdada pode assumir hipóteses sobre outras operações e atributos, os quais podem ter sidos alterados quando herdados.

Teste de interfaces – teste de componentes compostos que se concentra em ver se a interface de componente se comporta de acordo com sua especificação. Os casos de testes não são aplicados nos componentes individuais, mas sim na interface do componente composto. É importante para o desenvolvimento orientado a objetos. Os objetos e os componentes são definidos por suas interfaces e podem ser reusados causando erros.

Existem diferentes tipos de interfaces entre os componentes de programa:

Interfaces de parâmetros – dados ou referências a funções são passados de um componente para outro;

Interfaces de memória compartilhada – o bloco de memória é compartilhado entre os componentes. Os dados são colocados na memória por um subsistema e recuperados dali por outros subsistemas.

Interfaces de procedimentos – um componente engloba um conjunto de procedimentos que podem ser chamados por outros componentes. Os objetos e componentes reusáveis tem essa forma de interface.

Interfaces de passagem de mensagem – um componente solicita um serviço de um outro componente passando uma mensagem para ele. Uma mensagem de retorno inclui os resultados da execução do serviço. Alguns sistemas a objetos têm essa forma de interface, como ocorre em sistemas cliente-servidor.

Erros de interface são os mais comuns em sistemas complexos, veja suas categorais:

Mau uso de interface – um componente chama outro componente e comete um erro no uso de sua interface. Parâmetros do tipo incorreto, passados em ordem incorreta, quantidade incorreta.

Mau entendimento da interface – um componente chamador perde a especificação da interface do componente chamado e faz suposição sobre seu comportamento.

Erros de *timing* – ocorrem em sistemas de tempo real em que usa memória compartilhada ou uma interface de mensagem compartilhada. O consumidor pode acessar dados desatualizados porque o produtor das informações não atualizou as informações de interface compartilhada.

Este tipo de teste é difícil porque alguns defeitos de interface podem se manifestar somente sob condições incomuns.

Aqui estão algumas diretrizes:

1) Examine o código a ser testado e liste cada chamada a componentes externos. Projete um conjunto de testes em que os valores dos parâmetros dos componentes estejam no limite extremos de suas faixas.
2) Teste sempre a interface com ponteiros nulos onde os mesmos são passados por meio de uma interface.
3) Projete testes que causem a falha do componente onde este é chamado por meio de uma interface de procedimento.
4) Use o teste de estresse, em sistemas de passagem de mensagens. Projete testes que gerem muito mais mensagens do que podem ocorre na prática, problemas de timing podem ser revelados aqui.
5) Quando vários componentes interagem por meio de memória compartilhada, projete testes que variem a ordem na qual esses componentes são ativados. Estes podem revelar suposição implícitas feitas pelo programador a respeito da ordem na qual os dados compartilhados são produzidos e consumidos.

Técnicas de validação estáticas apresentam muitas vezes custos mais adequados do que os testes para descobrir erros de interface. Linguagem orientada a tipos como Java permite descobrir erros de interface.

A seguir apresentaremos uma síntese de projeto de casos de teste para as estratégias de teste de sistema e componentes:

Síntese - Um projeto de casos de teste visa projetar casos de entradas e saídas esperadas que testam o sistema.

Meta - criar um conjunto de casos de teste eficazes para descobrir defeitos do programa e demonstrar que o sistema atende aos requisitos.

Como fazer?

Selecione uma característica do sistema ou do componente que você vai testar.

Depois selecione um conjunto de entradas para executar aquelas características.

Documente as saídas esperadas.

Abordagens para projetar casos de teste:

Teste baseado em requisitos – testa os requisitos do sistema. Muito usada no estágio de teste de sistema quando são implementados vários componentes. Cada requisito é considerado e um conjunto de casos de teste é derivado para ele. É considerado um teste de validação.

Teste de partições – identifica partições de entradas e saídas e projetados testes de modo que todo sistema processe as entradas de todas as partições. Aqui envolve testes com classes de diferentes características como: números positivos, números negativos, seleções de menu. Assim são considerado partições de equivalência ou domínios (você espera que o programa se comporte da mesma maneira em relação a uma determinada classe de teste).

Partições de equivalência de entrada – são conjuntos de dados em que todos os membros do conjunto devem ser processados de modo equivalente. Entradas válidas e inválidas também formam partições equivalentes.

Partições de equivalência de saída – são saídas de programa com características em comum de maneira que elas possam ser consideradas como uma classe distinta.

Após identificar um conjunto de partições, você pode escolher casos de teste de cada uma dessas partições, uma boa regra é escolher casos de teste no limite das partições mais os casos próximos ao ponto médio da partição.

As partições podem ser identificadas por meio de especificação de programa ou documentação de usuário e também pela experiência que permite prevê valores de entradas com probabilidade de detectar erros.

Teste estrutural – é usado o conhecimento da estrutura do programa. Também conhecido como teste da caixa-branca. O entendimento do algoritmo usado em um componente pode ajudar a identificar partições e casos de teste adicionais.

Vetor de entrada (T)	Elemento chave (key)	Saída (Found, L)
17	17	Verdadeiro, 1
17	0	Falso, ?
12,18,21,23,32	23	Verdadeiro, 4

Tabela 2 Casos de teste para rotina de busca

Uma estratégia do teste estrutural é o teste de caminho, cujo objetivo é exercitar cada caminho independente, sendo que se todos os caminhos independentes forem executados pelo menos uma vez, todas as declarações no componente terão sido executadas pelo menos uma vez e todas as declarações condicionais são testadas para ambos os casos, verdadeira e falsa. O número de caminhos de um programa é proporcional a seu tamanho. Quando os módulos são integrados ao sistema, torna-se inviável usar técnicas de teste estrutural.

Vale comentar que o teste de caminho não testa todas as combinações possíveis de todos os caminhos do programa. Para quaisquer componentes, menos aqueles muito triviais sem loops, esse é um objetivo impossível.

O ponto de partida para o teste de caminho é um fluxograma do programa (modelo esquelético de todos os caminhos do programa). Consiste de nós que representam decisões e linhas que mostram o fluxo de controle. E cada ramo em uma declaração condicional (*IF-then-else* ou *case*) é mostrado como um caminho separado.

Tem como objetivo assegurar que cada caminho independente (aquele que atravessa pelo menos uma nova linha no fluxograma, exercitar uma ou mais condições novas) do programa seja executado pelo menos uma vez.

O cálculo da complexidade ciclomática ajuda a encontra o número de caminhos independentes. Nesta metodologia o número mínimo de casos de teste necessário para testar todos os caminhos é igual à complexidade ciclomática.

Analisadores dinâmicos são ferramentas de teste que trabalham em conjunto com os compiladores, que durante a compilação esses compiladores esses analisadores

adicionam instruções extras ao código gerado, contando o número de vezes em que cada declaração foi executada.

Ao projetar casos de teste devem começar com testes do mais alto nível com base nos requisitos e então progressivamente adicionar testes mais detalhados usando o teste de partição e estrutural.

Um outro teste importante que não é tratado por Pressman é o teste de usabilidade.

4.5 Teste de Usabilidade

Este tipo de teste envolve mais a área de desenho, projeto interface usuário, mas que deve ser revisado na fase de teste considerando utilização de manuais, *on-line help*, agentes e assistentes eletrônicos, etc.

Visa verificar a facilidade que o software possui de ser claramente entendido e facilmente operado pelos usuários. O teste aqui é baseado em *check-list* com vários itens que o software deve atender para que ele possa ser considerado para boa utilização.

Tendo como objetivos: **v**erificar a facilidade de operação do sistema pelo usuário, verificar a facilidade de entendimento das funções do sistema pelo usuário, através da utilização de manuais, *on-line help*, agentes e assistentes eletrônicos, etc.

Usado para executar diversas operações do sistema, utilizado a documentação do mesmo.

Um outro teste que não é abordado é o teste de conformidade.

4.6 Teste de Conformidade

Este teste verifica se a aplicação foi desenvolvida seguindo os padrões, procedimentos e guias da área de processos.

As metodologias são usadas para aumentar a probabilidade de sucesso do projeto, e portando devem ser testadas.

Tem como objetivos: **v**erificar se as metodologias de desenvolvimento de sistema estão sendo seguidas, garantir as conformidades aos padrões da empresa, avaliar se a documentação do sistema é racional e está completa.

Deve se usado quando se deseja que os padrões sejam seguidos, quando se deseja determinar o nível de seguimento dos padrões e quando se deseja identificar pontos falhos na metodologia.

A seguir faremos um breve relato do ponto de vista de Lan, considerando as seguintes estratégias: teste de sistemas e de componentes.

Para Lan o processo de teste de *software* nestas duas categorias tinham duas metas:

1) Demonstrar ao desenvolvedor e ao cliente que o *software* atende aos requisitos, significando que para software sob encomenda deve existir pelo menos um teste para cada requisito, para produtos genéricos deve existir testes para todas as características de sistema que serão incorporadas ao *release* do produto.
2) Descobrir falhas ou defeitos no *software* que apresentam comportamento incorreto, não desejável ou em não conformidade com sua especificação.

Ele considera o teste de aceitação dentro da primeira meta onde o sistema é executado corretamente em um dado conjunto de casos de teste que refletem o uso esperado do sistema, aqui um teste bem-sucedido é aquele em que o sistema funciona corretamente. Na segunda meta temos o teste de defeitos, casos de testes são projetados para expor defeitos aqui um teste bem-sucedido é o que expõe um defeito que causa o funcionamento incorreto do sistema.

Lan sugere também o planejamento da fase de teste, tendo como processo de planejamento o processo V&V. Os gerentes devem definir um responsável pelos estágios de teste. Sendo os desenvolvedores responsáveis por testar os componentes que desenvolveram. A tarefa é controlar a equipe para integrar os diversos módulos e construir o *software* e testar o sistema como um todo. Considerando sistemas críticos um processo mais formal pode ser usado, onde testadores independentes são responsáveis por todo estágio de teste.

Os testes de componente realizados por desenvolvedores baseiam-se no entendimento intuitivo de como os componentes devem operar. Já os testes de sistema baseiam-se em especificações detalhadas de requisitos do sistema ou uma especificação de características de alto nível orientada a usuários a serem implementadas no sistema.

Mediante as estratégias apresentadas anteriormente vale ressaltar a fase de depuração.

4.7 Depuração

Segundo Pressman, quando o caso de teste descobre um erro, a depuração é a ação que resulta na reparação do erro. A depuração não é teste, mas sempre ocorre em conseqüência do teste. A depuração tenta relacionar sintoma com causa levando assim a correção do erro. A mesma apresenta dois resultados: a causa será encontrada e corrigida ou a causa não será encontrada. No caso da última pode suspeitar de uma causa, projetar um caso de teste para ajudar a validar aquela suspeita e trabalhar para correção do erro de um modo interativo.

Independente da abordagem adotada, a depuração tem como objetivo primordial encontrar e corrigir a causa de um erro de *software.*

Em geral três estratégias de depuração foram proposta por [MYERS, 1979]:

Força Bruta – método mais comum e menos eficiente para isolar a causa de um erro de *software*. Adotamos a filosofia "deixe o computador encontrar o erro", são feitas listagem da memória, são invocados rastreadores da execução e o programa é carregado com comandos de saída.

Rastreamento – usada com sucesso em programas pequenos, o código-fonte é rastreado manualmente até que o lugar da causa é encontrado. Infelizmente, a medida que o número de linhas-fonte aumenta, o número de caminhos potenciais para trás pode se tornar inadmissivelmente grande.

Eliminação de causa – manifestada por indução ou dedução e introduz o conceito de particionamento binário. Os dados relacionados à ocorrência do erro são organizados para isolar causas em potencial. Sendo uma hipótese de causa concebida e os dados mencionados são usados para provar ou rejeitar a hipótese. De forma alternativa uma lista de todas as causas possíveis é desenvolvida e são conduzidos testes para eliminar cada uma. Se os testes iniciais indicam que uma hipótese particular de causa é promissora, os dados são refinados em uma tentativa de isolar o defeito.

A depuração automatizada é feita através de ferramentas que oferecem apoio semi-automático. Ambientes Integrados de Desenvolvimento (IDEs - *Integrated Development Environments*) fornecem um modo de captar alguns dos erros predeterminados específicos de linguagens. Uma grande variedade de compiladores de depuração, ajudas

dinâmicas de depuração (“rastreadores”), geradores automáticos de casos de teste e ferramentas de mapeamento de referência cruzada estão disponíveis. Porém ferramentas não são substitutos para avaliação cuidadosa baseada em um modelo completo de projeto de *software* e em um código-fonte claro.

A correção de um defeito pode introduzir outros erros. Van Vleck (VAN VLECK, 1989) sugere três perguntas simples que todo engenheiro de *software* deve formular antes de fazer a correção:

A causa do defeito está reproduzida em outra parte do programa? Verifique o padrão de lógica errado que muitas das vezes encontra em outro lugar que pode resultar na descoberta de outros erros.

Qual o próximo defeito que pode ser introduzido pelo concerto que estou prestes a fazer? Antes da correção deve-se avaliar o acoplamento da lógica e das estruturas de dados. Se a correção acontecer em uma parte altamente acoplada, cuidado especial deve ser considerado.

O que poderíamos ter feito para prevenir a ocorrência desse defeito? Move a garantia de qualidade de *software*. Se corrigirmos o processo, bem como o produto, o defeito será removido do programa em questão podendo ser eliminado de todos os futuros programas.

Neste tópico foi apresentada as estratégias de teste que podemos resumir sendo, os testes de unidade e de integração concentra-se na verificação funcional de um componente e na incorporação de componentes em uma estrutura de programa. Os testes de validação mostram a rastreabilidade aos requisitos de *software* e os testes de sistema validam o *software* depois de ter sido incorporado a um sistema maior.

Na seção seguinte abordaremos a questão do ambiente de teste e a importância da sua preparação.

5 AMBIENTES DE TESTES

Neste tópico será abordado o ambiente de teste.

Ambiente de teste é toda a infra-estrutura onde o teste será executado, compreendendo configurações de *hardware*, *software*, ferramentas de automação, equipe envolvida, aspectos organizacionais, suprimentos, rede e documentação. Sua finalidade é propiciar a realização de testes em condições conhecidas e controladas.

O ambiente de teste deve ser tratado na estratégia de teste e planejamento dos mesmos. O mesmo deve ser similar possível ao ambiente de teste que o usuário utilizará para o *software* em questão, sendo considerado assim como uma das fases mais difíceis, tendo como responsável o arquiteto de teste. Elementos como: massa de testes, modelos de dados, configuração dos *softwares* usados, tipo de teste, técnica de teste, também deverão ser considerados para a criação do ambiente de teste.

Veja a seguir os elementos do ambiente de testes que devem ser levados em consideração para o planejamento do ambiente de testes.

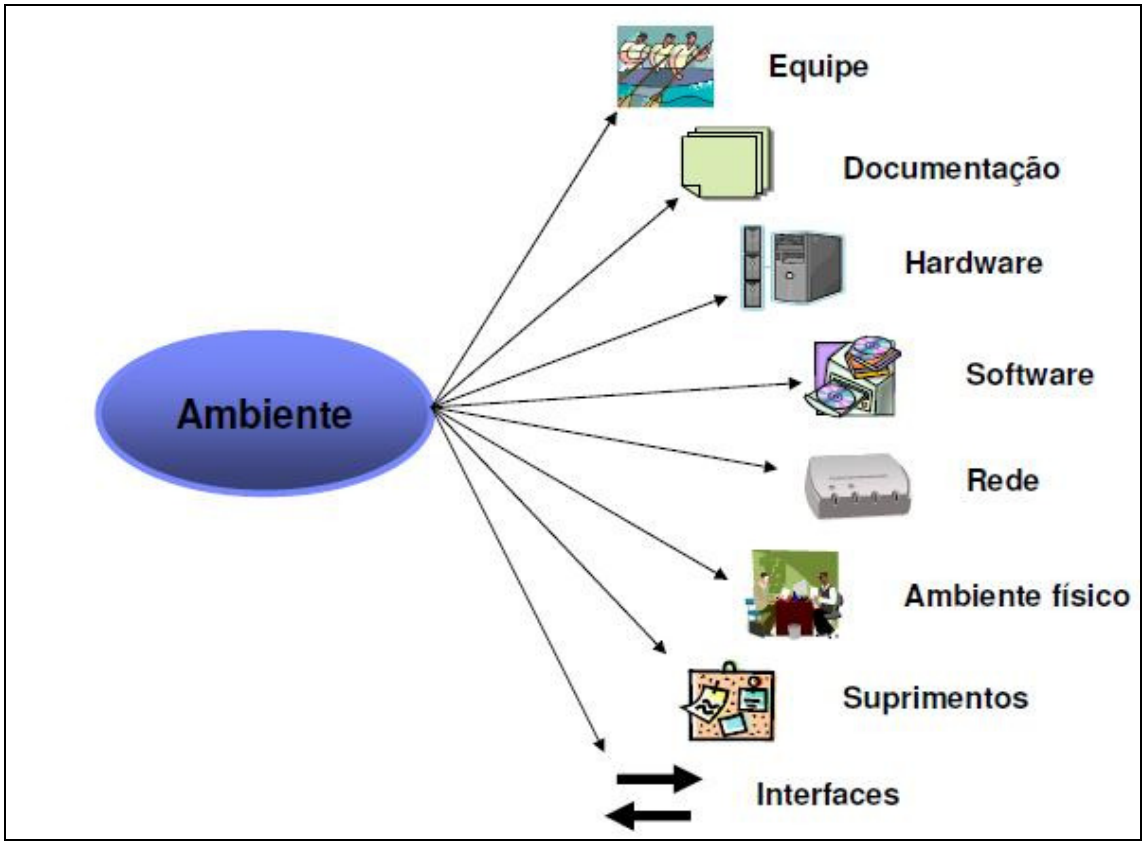


Figura 1 Visão macro do Ambiente de Teste (Kelvin Weiss, 14/03/10, página 3, Figura retirada do curso Fundamentos em Teste de Software, módulo 3)

5.1 Preparação do ambiente de teste

Alguns atributos do ambiente de testes precisam ser analisados e planejados para a realização dos testes. Veja na tabela abaixo a organização destes atributos de acordo com os tipos de teste:

Níveis de Teste	**Atributos**				
	Escopo	**Equipe**	***Volume de dados**	****Origem dos dados**	*****Interfaces e ambiente**
Testes Unitários	Unidade individual	Desenvolvedores	Volume pequeno	Criação manual	Não existem
Testes de Integração	Agrupamento de unidades	Desenvolvedores / Analistas de sistema	Volume pequeno	Criação manual	Não existem
Teste de sistema	Toda a aplicação	Analistas de sistema / Testadores	Volume grande	Criação automática e/ou dados reais	Simuladas / Reais
Aceitação	Toda a aplicação	Analistas de sistema / Testadores / Usuários	Volume grande	Dados reais	Reais

Tabela 3 Organização testes atributos de acordo com os tipos de teste – tabela elaborada conforme conteúdo do curso Fundamentos em Teste de Software, módulo 3.

* É a quantidade de dados utilizados durante o teste. Deve ser planejado a cada nível de teste.

- Testes de funcionalidade ou regressão
    - volume pequeno
- Testes de performance e carga
    - volume grande

** Depende diretamente do volume de dados.

Volume pequeno – criação manual, aqui importa a qualidade.

Volume grande – uso de ferramentas, aqui não importa a qualidade.

-Testes funcionais
    -utilizar dados reais

-Testes de performance e estresse

- não são necessários dados reais e sim o volume de dados.

*** Se refere se o *software* possui integração com outro software.

Os testes devem ser feitos dos dois lados, no *software* que sofreu a modificação e o que não sofreu modificação.

É importante trabalhar em ambientes de teste isolados para que não haja influências externa durante os testes de sistema e aceitação.

Determinar os testes relacionados ao ambiente de produção vai depender de vários fatores: tamanho do projeto, orçamento disponível e cronograma.

No processo de teste de *software* definimos três tipos de ambiente de teste:

Ambiente de Desenvolvimento compreende aos testes unitários e de integração.

Ambiente de Testes é composto pelo Teste de Integração, Sistema, Regressão, Aceitação e Funcionais.

Ambiente de Produção é composto pelo Teste de Estresse, Performance e de Aceitação.

5.2 Uso de ambientes virtuais

Na realidade atual, a maioria das empresas não prevê orçamento para preparação dos ambientes de testes na contratação de novos projetos de *software*.

Uma solução que vem ganhando espaço, por ser mais econômica, é a criação de ambientes virtuais (‘máquinas virtuais’).

Máquina virtual é um *software* que permite ao arquiteto de testes criar vários ambientes de testes, com diferentes configurações de *software*, *hardware*, sistemas operacionais, etc, utilizando na realidade a mesma máquina física.

5.3 Automação de Testes

A principio será abordado dois conceitos importantes na automação de testes:

Técnica – procedimento ou conjunto de procedimentos que tem como objetivo obter um determinado resultado.

Ferramenta – instrumento utilizado para aplicar a técnica.

Cada vez mais, ferramentas de automação de teste estão sendo lançadas no mercado para automatizar as atividades de teste. Mas antes de pensar em uma ferramenta é preciso definir os processos. Veja alguns fatores que leva a automação de testes:

_ Existem fortes pressões para melhorar a qualidade;

_ O projeto tem situações que não possam ser testadas adequadamente pelos métodos tradicionais,

_ O perfil dos softwares desenvolvidos forem complexos e com impacto no negócio,

_ Estudos de custo X benefício justificar o investimento,

_ O tamanho do projeto ou do ambiente de teste justificar.

A maioria das empresas sempre avalia a possibilidade do uso de ferramentas abertas e livres.

5.3.1. Tipo de Ferramentas

Existem basicamente três categorias de ferramentas de automação de testes, a saber:

5.3.1.1 Ferramentas de gerenciamento

As ferramentas de gerenciamento são as ferramentas gerenciais, normalmente utilizadas para fazer a gestão de testes e defeitos. Por exemplo, uma ferramenta que permite que sejam cadastrados os defeitos encontrados no *software* durantes os testes.

São ferramentas muito importantes, pois auxiliam o processo de teste como um todo.

Estão divididas em:

Ferramentas de gerenciamento de defeitos (rastreamento e correção dos defeitos): Jira[4], Mantis[5] (Free), BugZilla[6] (*Free*);

Ferramentas de controle de versionamento (documenta as versões dos testes): SubVersion[7], SourceSafe [8] (*Free*);

---

[4] Ferramenta desenvolvida pela Atlassian, comumente utilizada para rastreamento de erros, controle de problemas e gerenciamento de projetos.

[5] Ferramenta baseada na web que tem como principal função gerenciar defeitos de outros softwares.

[6] Ferramenta baseada em Web e e-mail que dá suporte ao desenvolvimento do projeto Mozilla, rastreando defeitos e servindo também como plataforma para pedidos de recursos.

Ferramenta de gerenciamento dos testes (gerenciam quais módulos deve ser testado, em qual data, dar suporte ao gerenciamento de testes e suas atividades. Faz a interface entre as ferramentas de execução, gerenciamento de defeito e gerenciamento de requisitos, gerar os resultados e os relatórios de progresso de teste): RTH, TestLink, Mantis

5.3.1.2 Ferramentas de verificação de código-fonte

Estas ferramentas são utilizadas para:

Verificar se o trabalho foi produzido dentro dos padrões de codificação;

Identificar pedaços de códigos não executados;

Identificar erros mais comuns, como problemas com inicialização de variáveis, estouro de memória, etc.

Estes tipos de ferramentas não têm a obrigação de verificar se o código realiza o que deveria realizar, ou seja, não têm a responsabilidade de testar as funcionalidades, sendo muito utilizada nos testes unitários.

5.3.1.3 Ferramentas de automatização na execução dos testes.

São ferramentas que auxiliam diretamente na execução dos testes. Veja alguns exemplos:

Unitários – Junit [9]

Sistema – TestComplet [10] (Teste Funcionais), JUnitPerf [11]

Aceitação – Jmeter [12](Teste Estresse), JUnitPerf (Teste de Performance).

---

[7] Ferramenta que veio substituir o CVS é um sistema de controle de versão que permite que se trabalhe com diversas versões de arquivos organizados em um diretório e localizados local ou remotamente, mantendo-se suas versões antigas e os logs de quem e quando manipulou os arquivos.

[8] Sistema de controle de versão em nível de arquivo que oferece funcionalidades de pontos de restauração e de coordenação de equipe.

[9] É um *framework open-source*, com suporte à criação de testes automatizados na linguagem de programação Java.

[10] Ferramenta utilizada para teste de software desenvolvida pela AutomatedQA.

[11] Extensão do JUnit, um conjunto de decoradores de testes JUnit, que é utilizado para medir o desempenho e a escalabilidade dos testes referenciados.

[12] Ferramenta utilizada para testes de carga em serviços oferecidos por sistemas computacionais. Esta ferramenta é parte do projeto Jakarta da Apache Software Foundation.

5.3.1 Objetivos da automação

Veja alguns dos objetivos na utilização da automação de teste:

Apoiar o processo de testes;
Reduzir falhas introduzidas pela intervenção humana;
Aumentar a produtividade (a médio/longo prazo);
Tratar a automação dos testes como um projeto.

A automatização deve acontecer quando:

Existirem fortes pressões para melhorar a qualidade;
O projeto tiver situações que não possam ser testadas adequadamente pelos métodos tradicionais;
O perfil dos *software*s desenvolvidos for complexo e com impacto no negócio;
Estudos de custo X benefício justificar o investimento;
O tamanho do projeto ou do ambiente de teste justificar.

Podemos concluir que uma inadequada infra-estrutura para testar *software*s leva à situação onde os defeitos são identificados mais tarde e que a garantia da integridade do ambiente de teste está diretamente relacionada à garantia de qualidade do produto.

A seguir iremos apontar diagnósticos dos riscos em projetos de testes.

6 DIAGNÓSTICOS DOS RISCOS EM PROJETOS DE TESTES

Nesta seção serão abordados diagnósticos dos riscos em projetos de testes. Antes serão apresentados alguns conceitos.

Um risco nunca é uma certeza de ocorrência. Se você sabe que uma coisa tem 100% de ocorrer isto não é um risco e sim um problema.

Os riscos podem ser tratados preventivamente ao contrário dos problemas. Podemos pensar em ações corretivas antes dos riscos se materializarem e virá um problema.

O que é risco?

É um evento ou condição incerta que, se ocorrer, provocará um efeito positivo ou negativo nos objetivos do projeto.

Exemplo de Risco:

Devido ao uso de uma nova tecnologia para a qual os programadores foram recém treinados, os mesmos poderão precisar de mais tempo para desenvolvimento dos programas, ocasionando um atraso nas atividades do cronograma.

Exemplo de Não-Risco:

Desconhecimento dos programadores quanto à tecnologia utilizada para a programação deste projeto.

Um risco pode concretizar um problema.

Em se tratando de projetos de teste de *software* podemos os categorizar em:

Riscos de projeto: são riscos ligados diretamente ao projeto, podemos citar como exemplo: requisitos, pessoal, recursos, clientes e cronogramas.

Riscos técnicos: são riscos relacionados à qualidade do *software* a ser desenvolvido.

Riscos para o negócio do cliente: são riscos relacionados a defeitos no *software*, que podem causar prejuízos ao negócio do cliente.

Sempre que aumentarmos o esforço dos testes, o risco de defeitos cairá. Porém, já vimos também que dificilmente podemos testar todas as funcionalidades que precisam ser testadas no prazo e no orçamento que dispomos. É necessário então fazer um levantamento dos módulos mais utilizados no momento do levantamento das estratégias de teste, feito isto e após a análise de riscos podemos identificar as prioridades para execução dos testes.

6.1 Maiores riscos de defeitos

Os maiores riscos de defeitos de um *software* estão em:

Funções muito complexas;

Funções freqüentemente alteradas;

Funções onde uma determinada ferramenta foi utilizada pela primeira vez no desenvolvimento;

Funções que foram construídas sob pressão;

Funções onde muitos defeitos já foram encontrados;

Funções com muitas interfaces.

Veja a baixo alguns exemplos de riscos de projetos como:

Ausência de um cronograma detalhado;

Dados dos testes não disponíveis na data programada;

Dados de testes ruins;

Disponibilidade da equipe de testes;

Disponibilidade de um ambiente de testes;

Falta de gerência de configuração.

Apontamos também alguns exemplos de riscos do processo:

Falta de gerência dos testes;

Falta de um processo de testes definido;

Falta de apoio da gerência para com a atividade de testes;

Falta de treinamento da equipe de testes;

Ambiente de testes inadequado.

6.2 Gerenciamento de riscos

Para Gerentes de Projeto, o Gerenciamento de Riscos é uma disciplina que busca pró-ativamente, minimizar os efeitos de eventos futuros que possam causar perdas para o projeto, ou maximizar os efeitos de eventos futuros que possam gerar ganhos para o projeto.

Assim podemos definir o gerenciamento de riscos como o processo de identificação, avaliação, priorização, desenvolvimento de estratégias de tratamento e acompanhamento dos riscos.

Conjunto de técnicas e ferramentas para identificar, estimar, avaliar, monitorar e administrar os acontecimentos que colocam em risco e execução do projeto.

A gerência de riscos possibilita uma melhoria continua no processo de tomada de decisão dentro da organização, pois dá a visibilidade a cerca das incertezas inerentes a um projeto de teste de softwares. Permitindo visualizar o que pode dar errado em nosso projeto.

São considerações importantes no gerenciamento de riscos:

Fornecer visibilidade acerca das incertezas inerentes a um projeto de teste de *software*;

Diminuir a tendência de otimismo extremo, de não querer enxergar os riscos ou simplesmente desejar para que eles não se materializem;

Gerar proteção contra as principais incertezas;

Toda gestão de projeto é gerenciamento de riscos.

Abordaremos os passos necessários para o gerenciamento de riscos, baseado no PMI.

Estes processos podem ser customizados de acordo com sua empresa. O gerenciamento de riscos deve ser executado em todo processo de teste de *Software*.

Identificar os riscos- identificar os eventos de riscos no projeto e quais as suas conseqüências. Podemos identificar os riscos das seguintes formas:

Avaliar problemas já enfrentados em outros projetos;

Reuniões de *brainstorm* com envolvidos;

Rever a documentação de *software*;

Utilizar uma lista de verificação com riscos conhecidos.

Analisar os riscos - Uma vez identificados os riscos, precisamos analisá-los a fim de encontrar o que os causam, e detalhar o seu impacto e a probabilidade da sua ocorrência.

Cada risco deve ser analisado sob a duas perspectivas:

Seu impacto – Prejuízo que um risco pode causar ao projeto;

Modelo de classificação: variação de 1 a 9.

Probabilidade de ocorrência - Chance de um risco se concretizar.

Modelo de classificação: variação de 1 a 9.

Priorizar e mapear os riscos – esta atividade tem como objetivo comparar os riscos e identificar quais são os mais importantes e que conseqüentemente merecem maior atenção.

A priorização é necessária porque normalmente os recursos do projeto são escassos, e devemos concentrar esforços primeiro nos riscos mais importantes.

Veja um exemplo de priorização de riscos:

Acompanhe abaixo uma matriz de riscos com sua priorização definida.

Cada risco deve ser analisado sob a perspectiva do seu impacto e probabilidade de ocorrência.

#	Fator de Risco	Prob.	Impacto	Criticidade
1	Risco 1	9	7	Alta
2	Risco 2	8	5	Alta
3	Risco 3	4	5	Média
4	Risco 4	1	2	Baixa

Tabela 4 Matriz de riscos com sua priorização definida.

A multiplicação da probabilidade e do impacto gera a criticidade.

Probabilidade / Impacto	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Criticidade Baixa

Criticidade Média

Criticidade Alta

Tabela 5 Matriz de riscos com sua priorização definida

Planejar resolução de riscos - uma vez priorizados e selecionados os riscos que serão atacados, para cada um deles deveremos preparar:

Um plano de mitigação – evita que um risco vire um problema.

O plano de mitigação pode ser considerado uma atividade pró-ativa, pois através dele tentamos minimizar o impacto e a probabilidade de um risco acontecer.

Suas características são:

Pró-ativo;

Contempla atividades para minimizar impacto e probabilidade de ocorrência;

Deve ser executado antes que o risco ocorra;

Evita que o risco se transforme em um problema;

Um plano de contingência.

Ao contrário do plano de mitigação, o plano de contingência é reativo, e deve ser acionado quando o risco virar um problema.

Suas características são:

Reatividade;

Sem chance de evitar que o risco ocorra.

Controlar e Monitorar os riscos – o objetivo desta atividade é acompanhar o progresso das ações planejadas e assegurar que elas sejam efetivas, além de monitorar as mudanças ocorridas nos riscos. O controle e monitoramento devem acontecer, pois:

Os riscos mudam durante o projeto;

Riscos podem surgir, outros podem desaparecer;

A criticidade dos riscos pode mudar.

Assim monitorar e controlar os riscos devem ser uma atividade contínua.

A maior parte dos riscos pode e deve ser tratada nas fases iniciais do projeto.

Mediante ao que foi exposto podemos fazer algumas considerações:

A gerência de riscos existe para que possamos nos precaver de futuros problemas

Podemos observar também que os riscos podem virar problemas e causar prejuízo ao projeto. Os planos de Mitigação e de Contingência devem ser elaborados e

seguidos a risca. Todos os envolvidos no projeto devem participar ativamente deste processo.

7 TÉCNICAS DE TESTES

Nesta seção discutiremos sobre as técnicas para os projetos de casos de teste de *software*, esta enfoca um conjunto de técnicas para a criação de casos de teste, que satisfazem aos objetivos globais e as estratégias de testes discutidas na seção 4 desta pesquisa.

7.1 Testes Caixa-Preta

Refere-se a testes que são conduzidos na interface do *software*, este examina algum aspecto fundamental do sistema, sem se preocupar com a estrutura lógica interna do *software*.

Também conhecido como teste comportamental, este tipo de teste tenta encontrar erros nas seguintes categorias:

Funções incorretas ou omitidas;

Erros de interface;

Erros de estrutura de dados ou de acesso a base de dados externa;

Erros de comportamento ou desempenho;

Erros de iniciação e término.

Aplicado durante os últimos estágios do teste, a atenção é focada no domínio da informação.

Esta aplicação deriva um conjunto de casos de teste que satisfazem os seguintes critérios: casos de teste que reduzem, de um valor que é maior do que 1, o número adicional de casos de teste que precisam ser projetados para atingir um teste razoável e casos de teste que nos dizem algo sobre a presença ou ausência de classes de erros, em vez de um erro associado somente com o teste especifico em mãos.

7.1.1 Métodos de Teste Baseado em grafo

Neste tipo de teste o primeiro passo é entender os objetos (objetos de dados, componentes tradicionais (módulos) e elementos orientados a objetos de *software* de computador) que estão modelados no *software* e as relações que conectam esses objetos, em seguida definir uma série de testes que verifica se "todos os objetos têm a relação esperada uns com os outros". Para isto o engenheiro de *software* cria um grafo, uma coleção de nós que representam objetos representados por círculos; ligações que representam as relações entre objetos, representadas por setas; pesos de nó que descrevem as propriedades de um nó. Com as seguintes ligações:

Ligação direcionada – uma seta indica que a relação se move em apenas uma direção.

Ligação bidirecional (ligação simétrica) – a relação é aplicada em ambas as direções.

Ligações paralelas – usadas quando diversas relações diferentes são estabelecidas entre nós de grafos.

Beizer (BEIZER, 1995) descreve alguns métodos de teste de comportamento que podem fazer o uso de grafos:

Modelagem de fluxo de transação – os nós representam passos em alguma transação (por exemplo, os passos necessários para fazer uma reserva em uma linha aérea usando o serviço on-line), use o diagrama de fluxo de dados (DFD) para criar grafos desse tipo.

Modelagem de estado finito – os nós representam diferentes estados do *software* observáveis pelo usuário (exemplo cada uma das telas abertas no momento que o usuário realiza uma operação), as ligações representam transições que ocorrem para ir de um estado para outro. Diagrama de Estado podem ser usados para representar esta modelagem.

Modelagem de fluxo de dados – nós são objetos de dados e as ligações são transformações que ocorrem para traduzir um objeto de dados em outro.

Modelagem de tempo - nós são objetos de programas e as ligações são conexões seqüências entre estes objetos. Os pesos das ligações são usados para especificar o tempo de execução necessário para o programa ser executado.

7.1.2 Particionamento de Equivalência

Este teste divide o domínio de entrada de um programa em classes de dados, das quais os casos de teste podem ser derivados. Um caso de teste ideal descobre sozinho uma classe de erros que poderia de outra forma exigir que muitos casos fossem executados antes que um erro geral fosse observado. Buscando definir um caso de teste que descobre classes de erros, reduzindo assim o número total de casos de testes que precisam ser criados.

O projeto de casos de teste para este tipo de teste é baseado em uma avaliação das classes de equivalência para uma condição de entrada, se o conjunto de objetos puder ser ligado por relações simétricas, transitivas e reflexivas, uma classe de equivalência estará presente. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada que é um valor numérico especifico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana.

Veja as diretrizes:

1 Se uma condição de entrada especifica um intervalo, uma classe de equivalência válida e duas inválidas devem ser definidas;

2 Se uma condição de entrada exige um valor específico, uma classe de equivalência válida e duas inválidas são definidas;

3 Se uma condição de entrada especifica o membro de um conjunto, uma classe de equivalência válida e uma inválida são definidas;

4 Se uma condição de entrada é booleana, uma classe de equivalência válida e uma inválida são definidas.

Veja um exemplo:

Dividir todas as combinações possíveis em classes. Em um sistema de gestão de contratos, a idade dos clientes varia de 18 a 120 anos.

Entrada	Valores Permitidos	Classes	Casos de Teste
Idade	Idade entre 18 e 120	18 a 120	Idade = 20
		< 18	Idade = 10
		> 120	Idade = 150

Tabela 6 Exemplo de Classe de Equivalência

7.1.3 Análise de valor limite

A análise de valor limite (BVA - boundary value analysis) leva a seleção de casos de teste que exercitam os valores limítrofes (erros que ocorre nas fronteiras do domínio de entrada) em vez de no "centro".

É uma técnica de projeto de casos de teste que completa o particionamento de equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, a BVA leva a seleção de casos de teste nas "bordas" da classe. Ao invés de focalizar somente as condições de entrada, ela deriva casos de teste também para o domínio de saída.

Veja as diretrizes:

1 Se uma condição de entrada especifica um intervalo limitado pelos valores a e b, casos de teste devem ser projetados com os valores a e b, e imediatamente acima e abaixo de a e b;

2 Se uma condição de entrada especifica vários valores, casos de teste devem ser desenvolvidos para exercitar os números mínimos e máximos. Valores imediatamente acima e abaixo devem ser testados;

3 Aplique as condições 1 e 2 às condições de saída. Casos de teste devem ser projetados para criar um relatório de saída que produza o número máximo (e mínimo) admissível de entradas na tabela;

4 Se as estruturas de dados internas do programa tem limites prescritos (exemplo: um vetor tem um limite definido de 100 entradas), elabore um caso de teste para exercitar a estrutura de dados no seu limite.

A maioria dos engenheiros de *software* realiza a BVA intuitiva em um certo grau. Aplicando estas diretrizes, o teste de limite será mais completo, tendo assim uma maior probabilidade de detecção de erros.

Veja um exemplo:

Entrada	Valores Permitidos	Classes	Casos de Teste
Idade	Idade entre 18 e 120	18 a 120	Idade = 18 Idade = 19 Idade = 119 Idade = 120
		< 18	Idade = 17
		> 120	Idade = 121

Tabela 7 Exemplo de Análise de Valor Limite

7.1.4 Teste de Matriz Ortogonal

Este tipo de teste é aplicado a problemas nos quais o domínio de entrada é relativamente pequeno, mas grande demais para acomodar o teste exaustivo, é útil para encontrar erros associados com falhas de regiões – uma categoria de erros associada com lógica defeituosa em um componente de *software*.

Exemplo: considere um sistema que tem três itens de entrada: X, Y e Z, cada um desses itens de entrada possui três valores discretos associados a eles. Há então 3^3 = 27 possíveis casos de teste.

Quando o teste de matriz ortogonal ocorre, é criada uma matriz ortogonal L9 de casos de teste, com a propriedade de balanceamento.

A avaliação dos resultados de um caso de teste utilizando a matriz L9 ocorre do seguinte modo:

Detecta e isola todas as falhas de modo singular – uma falha de modo singular é um problema consistente com qualquer nível de qualquer parâmetro singular, exemplo se todos os casos de teste do fator P1 = 1 causam um erro de condição, temos uma falha de modo singular. Pela análise da informação sobre quais testes revelam erros, pode-se

identificar quais valores de parâmetros causam a falha, este tipo de isolamento é importante para corrigir a falha.

Detecta todas as falhas de modo duplo – falha de modo duplo consiste se existe um problema consistente quando níveis específicos de dois parâmetros ocorrem simultaneamente. É uma indicação de incompatibilidade do par ou de interações danosas entre dois parâmetros de teste.

Falhas de multímodo – tais matrizes podem garantir a detecção apenas de falhas de modo singular e duplo. No entanto, muitas falhas de multímodo são também detectadas por esses testes.

Vale ressaltar que este tipo de teste permite projetar casos de teste que fornecem máxima cobertura com um número razoável de casos de teste, do que a estratégia exaustiva.

7.1.5 Testes Orientados a Objetos

A arquitetura Orientada a Objetos resulta em uma série de subsistemas em camadas que encapsulam classes de colaboração, sendo que cada uma dessas classes executam funções que ajudam a satisfazer aos requisitos do sistema.

É necessário testar um sistema OO em uma variedade de níveis diferentes para descobrir erros que podem ocorrer a medida que classes colaboram umas com as outras e subsistemas se comunicam entre as camadas arquiteturais.

Este tipo de teste é estrategicamente similar ao teste de sistemas convencionais, mas taticamente diferente.

Modelo de análise e projeto OO são similares na estrutura e no conteúdo para o programa OO, assim o "teste" pode começar com a revisão desses modelos. Após gerar o código, o teste OO real começa no "varejo" com testes elaborados para exercitar operações de classes e examinar se existem erros quando uma classe colabora com outras classes. Ao integrar estas classes formando um subsistema, o teste baseado no uso, com abordagens tolerantes a falhas, é aplicado para exercitar as classes que colaboram entre si.

Os casos de uso são usados para descobrir erros no nível de validação do *software*.

O projeto de caso de teste no método convencional aborda uma visão de software de entrada-processo-saída ou detalhes algorítimicos de módulos individuais, já teste orientado a objetos enfoca o projeto de seqüência apropriadas de operações para exercitar os estados de uma classe.

Como os atributos e operações são encapsulados, o teste de operações fora da classe é geralmente improdutivo. Apesar do encapsulamento ser um conceito de projeto essencial para OO, ele pode criar um pequeno obstáculo quando o teste é conduzido.

Vale ressaltar que a herança também leva a desafios adicionais para o projetista de casos de teste. Cada contexto de uso exige retestagem, mesmo que o reuso tenha sido concebido.

Se subclasses instanciadas de uma superclasse são usadas no mesmo domínio do problema, é provável que o conjunto de casos de teste derivado para a superclasse possa ser usado para testar a subclasse.

7.1.5.1 Aplicação de métodos convencionais de Projeto de Casos de Teste

Os métodos de teste caixa-branca podem ser aplicados a operações definidas para uma classe, no entanto a estrutura concisa de muitas operações de classe faz alguns argumentarem que o esforço aplicado a este tipo de teste poderia ser mais bem redirecionado para testes do nível da classe.

Já os métodos de teste caixa-preta são adequados para sistemas OO.

Veja os tipos de testes que contém a arquitetura OO:

7.1.5.2 Teste baseado em erro

Tem como objetivo projetar testes que tenham uma grande probabilidade de descobrir erros plausíveis, este tipo de teste começa com modelo de análise. Aqui casos de teste são projetados para exercitar o projeto ou o código. O sistema deve satisfazer aos requisitos do cliente, o planejamento pré-liminar para criação de testes baseados em erros começa com o modelo de análise.

O teste de integração procura erros plausíveis na chamada de operações ou nas conexões de mensagens, sendo que três tipos de erros são encontrados neste contexto:

resultado inesperado, uso da operação/mensagem errada e invocação incorreta. Este teste é aplicado tanto a atributos quanto a operações. A determinação de erros plausíveis deve ser feita quando funções (operações) são invocadas, o comportamento da operação deve ser examinado. O teste de integração tenta encontrar erros no objeto cliente, não no servidor, o foco é determinar se existem erros no código que chama, não no código chamado.

Este tipo de teste não encontra dois tipos principais de erros: especificações incorretas, ou seja, o produto não faz o que o cliente deseja, pode fazer a coisa errada ou pode omitir funcionalidade importante e ocorrem erros associados com a interação de subsistemas quando o comportamento de um subsistema cria circunstâncias (eventos, fluxo de dados) que provoquem a falha do outro subsistema.

Concluímos aqui que a efetividade dessa técnica depende de como os testadores consideram um erro plausível. Se erros reais em um sistema OO são considerados não plausíveis, então tal abordagem não é realmente melhor do que qualquer técnica aleatória. Se os modelos análise e projeto puderem fornecer conhecimento aprofundado sobre o que é provável dar errado, dessa forma o teste baseado em erros pode encontrar um número significativo de erros com esforço relativamente baixo.

7.1.5.3 Teste com base em cenário

O teste baseado em cenário concentra-se no que o usuário faz, não no que o produto faz, detecta as tarefas por meio de casos de uso, que o usuário precisa realizar depois aplicá-las, bem como suas variantes, aos testes. Vai descobrir erros que ocorrem quando qualquer ator interage com o *software*.

Cenários descobrem erros de interação, mas para isto os casos de teste precisam ser mais complexos e mais realísticos do que os testes baseados em erro.

Este tipo de teste exercita múltiplos subsistemas em um único teste, os usuários não se limitam ao uso de um subsistema de cada vez.

7.1.5.4 Teste da Estrutura Superficial e da Estrutura Profunda

A estrutura superficial refere-se à estrutura de um programa OO externamente observável, a estrutura que é imediatamente obvia a um usuário final, os testes são baseados nas tarefas do usuário, captar estas tarefas envolve entender, observar e falar com usuários representativos e não representativos.

Assim os melhores testes são originados quando o projetista olha o sistema de um modo novo ou não convencional. Aqui o projeto de casos de teste deve usar tanto os objetos quanto as operações como indícios que levem a tarefas despercebidas.

A estrutura profunda refere-se aos detalhes técnicos internos de um programa OO, a estrutura é entendida pelo exame do projeto e/ou código. Este tipo de teste é projetado para exercitar dependências, comportamentos e mecanismos de comunicação estabelecidos como parte do projeto do sistema de objetos do *software* OO. Aqui os modelos de análise e projeto são usados como base para o teste de estrutura profunda. O diagrama de colaboração UML ou o modelo de implantação mostram as colaborações entre objetos e subsitemas que podem não ser externamente visíveis.

7.1.5.5 Métodos de Teste Aplicáveis ao nível de classe

O teste aleatório e de partição são métodos que podem ser usados para exercitar uma classe durante o teste OO.

7.1.5.5.1 Teste Aleatório para classe OO

Para cada classe defini as possíveis operações da mesma aplicando algumas restrições de acordo com a natureza do problema, mesmo com tais restrições existe muitas permutações das operações. Gerando assim uma variedade de diferentes seqüências de operação que pode ser gerada aleatoriamente.

Para este tipo de teste o número de permutações possíveis pode ficar muito grande, aplicar o teste de matriz ortogonal pode ajudar a melhorar a eficiência deste teste.

7.1.5.5.2 Teste de Partição no nível de classe

Este tipo de teste reduz o número de casos de teste necessários para exercitar a classe de um modo semelhante ao da partição de equivalência para *software* validar se o sistema atende aos requisitos e assegurar que o sistema é confiável. Entrada e saída são categorizadas e casos de testes são projetados para exercitar cada categoria. Veja as categorias de partição:

Partição baseada em estado – categoriza as operações de classes com base na sua capacidade de mudar o estado da classe. Os testes são projetados de modo que exercitem, separadamente as operações que mudam o estado e aquelas que não mudam o estado.

Partição baseada em atributo - categoriza as operações de classes com base nos atributos que elas usam que podem ser divididas em: operações que usam, operações que modificam e operações que não usam e nem modificam.

Seqüências de teste são então projetadas para cada partição.

Partição baseada em categoria - categoriza as operações de classes com base na função genérica que cada uma realiza.

7.1.5.6 Projeto de casos de Teste Interclasse

Quando a integração do sistema OO tem início o projeto de casos de teste torna-se mais complicado. É nesse estágio que deve começar o teste das colaborações entre classes.

Como o teste das classes individuais, o teste de colaboração de classes pode ser conseguido aplicando-se os métodos aleatórios e de particionamento, bem como o teste baseado em cenário e teste comportamental.

7.1.5.6.1 Teste de várias classes

A abordagem para o teste de partição de várias classes é semelhante à abordagem usada para o teste de partição de classes individuais, sendo uma única classe particionada. A seqüência de teste são expandidas para incluir aquelas operações invocadas por meio de mensagens para as classes colaboradoras, onde uma abordagem alternativa particiona os testes com base nas interfaces de uma classe particular.

Segue abaixo uma seqüência de passos para gerar casos de teste aleatórios para várias classes:

Para cada classe cliente, use a lista de operações de classe para gerar uma série de seqüências de teste aleatório. As operações vão enviar mensagens para outras classes servidoras.

Para cada mensagem gerada, determine a classe colaboradora e a operação correspondente no objeto servidor.

Para cada operação no objeto servidor (que foi invocado por mensagens enviadas pelo objeto cliente), determine as mensagens que ele transmite.

Para cada uma das mensagens, determine o nível seguinte de operações que são invocadas e incorpore-as à seqüência de teste.

7.1.5.6.2 Testes Derivados dos Modelos de Comportamento

O diagrama de estado de uma classe pode ser usado para ajudar a originar as seqüências de testes que vão exercitar o comportamento dinâmico da classe e daquelas classes que colaboram com ela.

Os testes a ser projetados devem cobrir todos os estados, ou seja, as seqüências de operações devem fazer a classe realizar transição entre todos os estados permitidos.

Em situações nas quais o comportamento da classe resulta em uma colaboração com uma ou mais classes, diagramas de estado múltiplos são usados para rastrear o fluxo de comportamento do sistema.

O modelo de estados pode ser percorrido em forma de “inclusão progressiva” neste contexto implica um caso de teste exercitar uma única transição e, quando uma nova transição tiver de ser testada, são usadas apenas aquelas previamente testadas.

7.1.6 Teste de ambientes, arquiteturas e aplicações especializadas

A seguir serão abordadas diretrizes para teste de ambientes, arquiteturas e aplicações especializadas comumente encontradas por engenheiros de *software*.

7.1.6.1 Teste de IGU

Interfaces gráficas com o usuário (IGU) ou *Graphical User Interfaces* (GUI), mostra desafios interessantes para os engenheiros de software. O uso de componentes reusáveis como parte de ambientes de desenvolvimento IGU, a criação de interface com o usuário tornou-se menos demorada e precisa, mas ao mesmo tempo, a complexidade das IGU cresceu, levando a dificuldades no projeto e na execução de casos de teste.

Aqui grafos de modelagem de estados finitos podem ser usados para derivar uma série de testes que tratam de objetos de dados e programa específicos, relevantes para a IGU. Este tipo de teste deve ser conduzido usando-se ferramentas automatizadas.

7.1.6.2 Teste de Arquitetura Cliente/Servidor

Este tipo de teste é um desafio para os testadores de *software*. A natureza distribuída de ambientes cliente/servidor, os aspectos de desempenho, associados com processamento de transações, a presença potencial de várias plataformas de *hardware* diferentes, as complexidades de redes de comunicação, a necessidade de atender a muitos clientes por uma base de dados centralizada (ou distribuída) e os requisitos de coordenação impostos ao servidor são indicativos que torna difícil o teste de *software* do que em aplicações isoladas.

O teste cliente/servidor ocorre em três níveis:

Aplicações clientes individuais são testadas no modo “não conectado”, a operação do servidor e a rede subjacente não são consideradas;

O *software* cliente e as aplicações do servidor associadas são testadas em conjunto, mas as operações da rede não são explicitamente exercitadas;

A arquitetura completa cliente/servidor, incluindo operações e desempenho da rede, é testada.

As seguintes abordagens de teste são comumente encontradas para aplicações cliente/servidor:

Teste de função da aplicação – a aplicação é testada de modo isolado.

Teste de servidor – o desempenho do servidor (tempo de resposta global e vazão de dados (*throughpu*) é também considerado.

Teste de banco de dados – a precisão e a integridade dos dados armazenados pelo servidor são testadas. As transações são examinadas para garantir que os dados sejam armazenados, atualizados, recuperados e arquivados adequadamente.

Teste de transação – uma série de testes é criada para garantir que cada classe de transações seja processada de acordo com os requisitos. Podemos abordar aqui tempo de processamento e volume de transação.

Teste de comunicação em rede – verifica se a comunicação entre os nós da rede ocorre corretamente e se a passagem de mensagens, transações e tráfego de rede relacionado ocorre sem erro. Testes de segurança da rede podem também ser conduzidos como parte desses testes.

Recomenda-se o desenvolvimento de perfis operacionais derivados de cenários de uso cliente/servidor. Um perfil operacional indica como diferentes tipos de usuário interoperam com o sistema cliente/servidor, os perfis fornecem um “padrão de uso” que pode ser aplicado quando testes são projetados e executados.

7.1.6.3 Teste da Documentação e Dispositivos de Ajuda

Erros na documentação pode ser tão grave para a aceitação do programa quanto erros nos dados ou código-fonte. Nada é mais frustrante que seguir exatamente um guia do usuário ou um documento de ajuda *on-line* e obter resultados ou comportamentos

que não coincidem com aqueles previstos pela documentação. Assim o teste de documentação deve ser uma parte significativa de todo o plano de teste de *software*.

Este teste pode ser abordado em duas fases: revisão e inspeção examina o documento quanto a clareza editorial e teste ao vivo usa a documentação em conjunto com o uso do programa real.

7.1.6.4 Teste de Sistemas de Tempo Real

O motivo que nos leva a este tipo de teste é que em muitas situações, dados de testes fornecidos vão resultar em processamento adequado, quando um sistema de tempo real está em um estado, enquanto os mesmos dados fornecidos quando o sistema está em um estado diferente podem levar a erro.

Além do mais teste de *software* devem considerar o impacto de falhas do *hardware* no processamento de *software*. Estas falhas podem ser extremamente difíceis de simular com realismo.

Segue abaixo uma estratégia de quatro passos:

Teste de tarefa – testa cada tarefa independente. Os testes convencionais são projetados e executados para cada tarefa, detectando assim erros de lógica e de função.

Teste comportamental – examina seu comportamento como conseqüência de eventos externos, com o uso de ferramentas automatizadas. Este tipo de teste pode servir de base para projetos de casos de teste usados após a construção do *software*.

Teste de inter-tarefas – tarefas assíncronas que se comunicam umas com as outras são testadas com diferentes taxas de dados e carga de processamento para detectar se erros de sincronização inter-tarefas vão ocorrer.

Teste de sistema – descobrir erros na interface *software*/*hardware*. Com uso do diagrama de estados e a especificação de controle, o testador desenvolve uma lista de todas as possíveis interrupções e do processamento que ocorre em conseqüência das interrupções, projetando testes para avaliar as seguintes características do sistema:

- Se as prioridades de interrupção estão atribuídas adequadamente e propriamente manipuladas.

- O processamento para cada interrupção foi conduzido corretamente.

- Verificar o desempenho (tempo de processamento) de cada procedimento de manipulação de interrupção se está de acordo com os requisitos.

- Se um alto volume de interrupções chegando em tempos críticos cria problemas de função ou desempenho.

Áreas globais de dados, usadas para transferir informação no processamento de interrupções, devem ser testadas para avaliar o potencial de geração de efeitos colaterais.

Fechamos aqui os métodos que compreende a técnica de testes de caixa-preta, abordaremos a seguir os métodos aplicados nos testes caixa-branca.

7.2 Testes Caixa-Branca ou Teste Caixa de Vidro

Baseado em um exame rigoroso do detalhe procedimental. Caminhos lógicos internos ao *software* e colaborações entre componentes são testados definindo-se casos de testes que exercitam conjuntos específicos de condições e/ou ciclos. A forma exaustiva de definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, não pode levar a descartá-lo, um número limitado de caminhos lógicos importantes podem ser selecionado e exercitado, estruturas de dados também podem ser submetidas à prova quanto à validade.

Conhecido também como teste caixa de vidro ou teste estrutural, nos casos de teste usa a estrutura de controle descrita como parte do projeto ao nível de componentes. Neste contexto derivam-se os seguintes casos de teste: Garantir que todos os caminhos independentes de um módulo tenham sido exercitados pelo menos uma vez, exercitar todas as decisões lógicas em seus lados verdadeiros e falsos, executam todos os ciclos nos seus limites e dentro de seus intervalos operacionais, exercitarem as estruturas de dados internas para garantir sua validade.

Segue abaixo técnicas de Teste de Caixa-Branca:

Os testes a seguir são considerados teste de Estrutura de Controle.

7.2.1 Teste de Caminho Básico

Este tipo de teste permite ao projetista de casos de testes originar uma medida da complexidade lógica de um projeto sentimental e usar essa medida como guia para

definir um conjunto básico de caminhos de execução, estes executam com garantia cada comando do programa pelo menos uma vez durante o teste.

7.2.1.1 Notação de Grafo de Fluxo

Este grafo mostra o fluxo de controle lógico, cada circulo chamado de nó do grafo de fluxo representa um ou mais comandos procedimentais. As setas chamadas de arestas ou ligações representam o fluxo de controle e são análogas às setas de fluxograma. As áreas limitadas por arestas e nós são chamadas de regiões. Vale ressaltar que um grafo de fluxo deve ser desenhado somente quando a estrutura lógica de controle de um componente é complexa, permitindo seguir mais facilmente os caminhos do programa.

O nó que contém uma condição é chamado de nó predicado e é caracterizado por duas ou mais arestas saindo dele.

7.2.1.2 Caminhos independentes de programa

É qualquer caminho ao longo do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição, este caminho deve pelo menos incluir uma aresta que não tenha sido atravessada antes do caminho ser definido. O mesmo é obtido mediante o grafo de fluxo.

Os caminhos constituem o conjunto-base, onde testes são projetados para forçar a execução desses caminhos, onde todo comando do programa terá sido executado pelo menos uma vez e cada condição executada no seu lado verdadeiro e falso.

Complexidade ciclomática – este cálculo mostra quantos caminhos deve-se procurar. É uma métrica de *software* que fornece uma medida quantitativa da complexidade lógica de um programa.

A complexidade ciclomática tem sua fundamentação na teoria dos grafos e pode ser calculada de três maneiras:

1. O número de regiões corresponde a complexidade ciclomática;

2. A complexidade ciclomática V(G), para um grafo de fluxo G, é definida como:

V(G) = E – N + 2

E: número de arestas (E-edges)

N: número de nós (N-nodes)

3. Pode ser definida como:

V(G) = P + 1

P: número de nós predicados (P-Predicate nodes).

Esta métrica pode ser útil tanto para planejamento de teste quanto para projeto de casos de teste. Fornecendo o limite superior de número de casos de teste que precisam ser executados, para garantir que cada comando de um componente tenha sido executado pelo menos uma vez.

7.2.1.3 Método de Projeto de Casos de Teste

O método de teste do caminho básico pode ser aplicado a um projeto procedimental ou a um código fonte.

Os seguintes passos podem ser aplicados para originar o conjunto-base:

1. Usando o projeto ou código como base, desenhe o grafo de fluxo correspondente. Usando os símbolos e regras de construção apresentados na seção 7.2.1.2. A criação do grafo se dá enumerando os comandos PDL que serão mapeados nos nós correspondentes do grafo;

2. Determine a complexidade ciclomática do grafo de fluxo resultante. A complexidade pode ser determinada sem desenvolver um grafo de fluxo através da contagem de todos os comandos condicionais na PDL;

3. Determine um conjunto-base de caminhos linearmente independentes. Os nós predicados ajudam na definição de casos de teste.

4. Prepare casos de teste que vão forçar a execução de cada caminho de conjunto-base.

Cada caso de teste é executado e comparado aos resultados esperados, uma vez completados todos os casos, o testador pode ter certeza que todos os comandos do programa foram executados pelo menos uma vez.

É importante observar que alguns caminhos independentes não podem ser testados de um modo individual, nestes casos esses caminhos são testados como parte de outro teste de caminho.

7.2.1.4 Matriz de grafo

Para originalizar o grafo de fluxo e determinar um conjunto de caminhos básicos através de mecanização através de uma ferramenta é usada uma estrutura de dados chamada matriz de grafo (é uma matriz quadrada onde o número de linhas e de colunas é igual ao número de nós do grafo de fluxo. Onde cada linha e coluna corresponde a um nó identificado, e as entradas na matriz correspondem as conexões (aresta) entre nós.
Adicionando um peso de ligação a cada entrada da matriz, esta pode tornar uma possante ferramenta para avaliar a estrutura de controle, fornecendo informação adicional sobre o fluxo de controle. Assim temos o peso de ligação 1 (existe uma conexão) ou 0 (não existe uma conexão), estes mesmos pesos podem ser atribuídos a outras propriedades mais interessantes como:
- A propriedade de uma aresta será executada.
- O tempo de processamento gasto durante o percurso de uma ligação.
- A memória necessária durante o percurso de uma ligação.
- Os recursos necessários durante o percurso de uma ligação.

E um rigoroso tratamento de outros algoritmos matemáticos que podem ser aplicados a matrizes. Com estas técnicas, a análise necessária para projetar casos de teste pode ser parcial ou totalmente automatizada.

Segue abaixo outras variações do teste da estrutura de controle, elas ampliam a cobertura de teste e melhora a qualidade de teste caixa-branca.

7.2.2 Teste de Estrutura de Controle

7.2.2.1 Teste de Condição

Este tipo de teste exercita as condições lógicas contidas em um módulo de programa, onde podemos ter uma:

Condição simples – uma variável booleana ou uma expressão relacional, possivelmente precedida por um operador Não.

Condição Composta – formada de duas ou mais condições simples, operadores booleanos e parênteses.

Uma condição sem expressões relacionais é referida como expressão booleana.

Dessa forma os tipos de elementos possíveis em uma condição booleana incluem um operador booleano (OU, E, NÃO), uma variável booleana, um par de parênteses (envolvendo uma condição simples ou composta), um operador operacional ou uma expressão aritmética.

Sendo que se uma condição está incorreta, pelo menos um componente da condição está incorreto. Assim os tipos de erros em uma condição se referem aos elementos citados anteriormente. Este tipo de teste focaliza o teste de cada condição do programa para garantir que não contém erros.

7.2.2.2 Teste de Fluxo de Dados

Este tipo de teste seleciona caminhos de teste e um programa de acordo com a localização das definições e dos usos das variáveis no programa, onde a cada comando de um programa é atribuído um número de comando único e que cada função não modifica seus parâmetros ou variáveis globais.

7.2.2.3 Teste de Ciclo (loops)

Ciclos considerados como pedra fundamental da grande maioria de todos algoritmos implementados em software, porém freqüentemente é dado a eles pouca atenção na fase de testes.

Este teste focaliza exclusivamente a validade de construções de ciclo, aqui quatro diferentes classes de ciclos: ciclos simples, concatenados, aninhados e desestruturados.

Ciclos simples – este conjunto de testes pode ser aplicado a ciclos simples em que n é o número máximo de passagens permitidas pelo ciclo.

Ciclos aninhados – neste caso o número de testes possíveis cresceria geometricamente, à medida que o nível de aninhamento crescesse, resultando em um número de testes impraticável. Existe uma abordagem que ajuda a reduzir este número de teste:

1. Comece no ciclo mais interno, ajustando todos os outros ciclos para os valores mínimos.
2. Conduza testes de ciclo simples para o ciclo mais interno enquanto mantém os ciclos externos nos seus valores mínimos do parâmetro de iteração (exemplo, contador de ciclo). Adicione outros testes para os valores fora do intervalo ou excluídos.
3. Trabalhe em direção ao exterior, conduzindo testes para o ciclo seguinte, mas mantendo todos os outros ciclos externos nos valores mínimos e os outros ciclos aninhados em valores “típicos”.
4. Continue até que todos os ciclos tenham sido testados.

Ciclos concatenados – estes podem ser testados usando a abordagem definida para ciclos simples, se cada um dos ciclos é independente do outro. Se dois ciclos são concatenados e o contador de ciclo, para o ciclo 1, é usado como valor inicial para o ciclo 2, então os ciclos não são independentes, dessa forma a abordagem aplicada a ciclos aninhados é recomendada.

Ciclos desestruturados – essa classe de ciclos deve ser reprojetada para refleti o uso de construções de programação estruturada.

7.3 Padrões de teste

Aproveitamos esta seção para falarmos um pouco sobre padrões de teste.

Pressman cita padrões de teste mediante definições de Marick (MARICK, 2002).

Padrões de teste é um mecanismos para descrever blocos construtivos de *software* ou situações de engenharia de *software*, que são obtidos à medida que diferentes aplicações são construídas ou projetos são conduzidos. São situações encontradas que testadores de *software* podem conseguir reusar quando eles abordam o teste de algum sistema novo ou revisado.

Benefícios:

Fornecem vocabulário para resolver problemas.

Dá atenção nas influências por trás do problema, onde projetistas são capazes de entender melhor quando e porque uma solução se aplica.

Incentiva o raciocínio interativo. Onde cada solução cria um contexto no qual novos problemas são resolvidos.

Concluísse que padrões de teste podem ajudar uma equipe de *software* a se comunicar mais efetivamente sobre testes, entender as influências motivadoras que levam a uma abordagem especifica de teste e abordam o projeto de casos de teste como uma atividade evolutiva.

A seguir abordaremos o processo V&V.

8 V&V (VERIFICAÇÃO E VALIDAÇÃO)

Será abordado aqui o processo V&V do ponto de vista de LAN SOMERVILLE, no que se diz respeito a seu planejamento, a relação com a inspeção de *software*, a análise estática automatizada e por fim os métodos formais utilizados.

São duas grandes fases do processo de teste. Durante e depois do processo de implementação, o programa em desenvolvimento deve ser verificado para certificar-se de que ele atende a sua especificação e entrega a funcionalidade esperada. Verificação e Validação (V&V) é denominação dada a esses processos de verificação e análise. Estas atividades ocorrem em cada estágio do processo de *software*, começando com as

revisões de requisitos e continua ao longo das revisões de projeto e das inspeções de código até o teste de produto.

Verificação - envolve verificar se o *software* esta de acordo com as especificações se atende aos requisitos funcionais e não-funcionais especificados.
Permitindo encontrar defeitos na fase inicial do projeto de desenvolvimento, realizada através de revisões e inspeções técnicas em documentos e códigos do projeto.
É realizada com auxilio de *check list* com item de qualidades que um produto deve ter.

Validação – assegura que o sistema atende às expectativas do cliente, vai além de verificar se o sistema está conforme a sua especificação para mostrar que o *software* realiza o que o cliente espera que ele faça.
Executadas através de testes como: funcional, regressão, de desempenho.
Estes dois testes são complementares e executados durante todo ciclo de vida do processo de testes. Os testes de verificação são executados antes dos testes de validação.

8.1 Objetivo

Estabelecer confiança de que o sistema de *software* esta adequado a seu propósito. O nível de confiabilidade vai depender do propósito do sistema, das expectativas dos usuários do sistema e do atual ambiente de mercado.

8.2 Abordagens

Dentro do processo V&V existe duas abordagens complementares para a verificação e análise de sistema:

Inspeções de *Software* ou revisões por pares analisam e verificam representações de sistemas como documento de requisitos, diagramas de projeto e código-fonte do programa. Elas podem ser suplementadas por alguma análise automática de texto-fonte de um sistema ou de documentos associados.

Testes de *software* executa uma implementação do *software* com dados de teste. É examinada as saídas do *software* e seu comportamento operacional para verificar se seu desempenho está conforme necessário. È uma técnica dinâmica do processo V&V.

As inspeções de *software* podem ser usadas em todos os processos de *software*. Revisões de requisitos e de projetos são as principais técnicas usadas para detecção de erros na especificação e no projeto.

Tais técnicas incluem inspeções de programas, análise de código-fonte automatizada e verificação formal (verificação), elas não podem demonstrar que o *software* é útil operacionalmente e nem verificar propriedades emergentes do *software* como desempenho e confiabilidade.

Apesar das inspeções de *software* ser usadas, o teste de programa será sempre a técnica principal de verificação e validação de *software*. Teste envolve informar dados reais processados pelo programa retornando saídas válidas ou não. Existem dois tipos de teste usados neste estágio:

Teste de validação – tem a finalidade de mostrar que o *software* é o que o cliente deseja. Podendo usar testes estatísticos para testar desempenho e a confiabilidade do programa e para verificar sua funcionalidade sob condições operacionais.

Teste de defeitos – revela defeitos no sistema em vez de simular o seu uso operacional. Seu objetivo é encontrar inconsistências entre um programa e sua especificação.

O processo V&V e de *debugging* são intercalados, ou seja, à medida que descobre defeitos no programa, deve-se alterar o programa para corrigir esses defeitos. Não existe métodos simples para o processo de *debugging*, algumas pessoas experientes procuram por padrões nas saídas dos testes aonde o defeito se manifesta e usam o conhecimento sobre o tipo de defeito, o padrão de saída, a linguagem de programação e o processo de programação para localizar o mesmo.

Assim será necessário rastrear linha por linha. Ferramentas de *debugging* que coletam informações sobre a execução do programa também podem ajudar a localizar a origem de um problema, elas fazem parte de um conjunto de ferramentas de apoio da linguagem que estão integradas com um sistema de compilação.

Depois de corrigido o defeito encontrado o mesmo deve passar por uma nova inspeção ou teste de regressão, tal teste é usado para verificar se as novas mudanças feitas em um programa não introduziram novos defeitos. A realidade vem mostrando

que uma alta proporção de reparo de defeitos é incompleta e introduz novos defeitos no programa.

O plano de teste tende a facilitar tal trabalho, se neste plano for identificado dependências entre componentes e os testes associados a cada componente, existindo assim rastreabilidade dos casos de testes aos componentes testados. Com esta rastreabilidade documentada, será possível executar um subconjunto de casos de testes para verificar o componentes modificado e seus dependentes.

8.3 Planejamento

Dependendo do sistema a se aplicar o processo V&V mais da metade do orçamento de desenvolvimento do sistema é gasto aqui. No planejamento V&V deve-se especificar padrões e procedimentos para inspeções e testes de *software*, estabelecer checklists para orientar as inspeções de programa.

Os planos de teste são destinados aos engenheiros de *software* envolvidos a projetar e realizar os testes de sistemas. Este define os recursos de hardware e de *software* necessários. Devem incluir quantidades significativas de contingência de modo que enganos no projeto e na implementação possam ser acomodados e o pessoal possa ser realocados a outras atividades. Não são documentos estáticos, evoluem durante o processo de desenvolvimento, eles mudam devido a atrasos em outros estágios do processo de desenvolvimento. Se parte de um sistema está incompleta o sistema como um todo não pode ser testado.

8.3 O modelo “V”

A estrutura do modelo V é uma aproximação do processo de testes que pode ser integrada com todo a processo de desenvolvimento. Este modelo representa o desenvolvimento versus o teste.

Este focaliza em testar durante todo o ciclo de desenvolvimento para conseguir uma detecção adiantada dos erros.

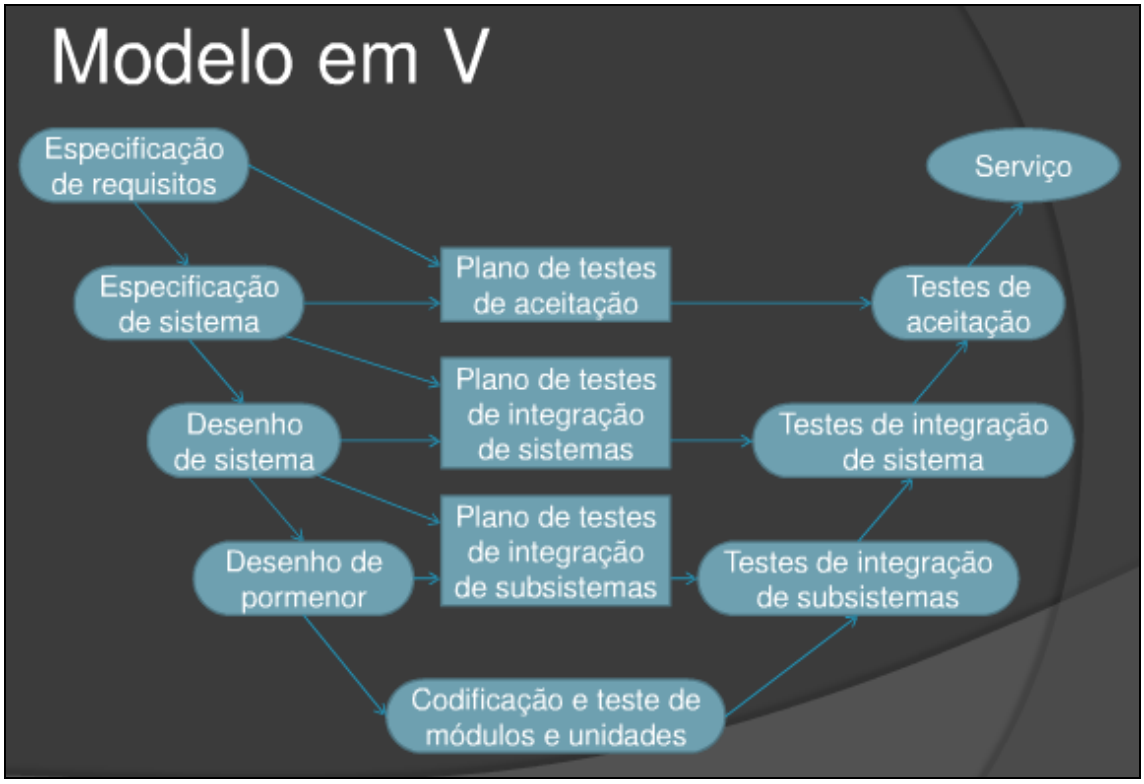


Figura 2 Modelo V (Manuel Siqueira de Menezes, 15/-5/10 - Figura extraída de: http://www.slideshare.net/MMSequeira/verificao-e-validao)

A Parte esquerda do V mostra as etapas do desenvolvimento de software e a parte direita mostra as etapas de testes (níveis).

O Processo V focaliza em testar em todo nível de desenvolvimento.

Ao final de cada etapa do processo de desenvolvimento o produto (saídas das atividades de desenvolvimento) é avaliado, verificado e validados.

Este modelo retrata a importância do teste de *software* no inicio do ciclo de desenvolvimento e garante a qualidade do mesmo porque os produtos são testados varias vezes ao longo do ciclo de vida mostrando que o teste não é uma etapa uma fase, mas um processo integrado ao ciclo de desenvolvimento.

8.4 Inspeções de *Software*

É o processo V&V estático, no qual um sistema de *software* é revisto para encontrar erros, omissões e anomalias, As inspeções enfocam o código-fonte, mas qualquer representação legível do *software* como seus requisitos ou um modelo de projeto, pode ser também inspecionado. Quando se inspeciona um sistema, é usado o

conhecimento do sistema, seu domínio de aplicação e a linguagem de programação ou o modelo de projeto para descobrir erros.

Vantagens da inspeção sobre o teste:

Sendo a inspeção um processo estático não é preciso preocupar com interações, com isso uma única sessão de inspeção pode descobrir muitos erros de um sistema.

Versões incompletas de um sistema podem ser inspecionadas sem custos adicionais. Se um sistema está incompleto é necessário desenvolver conjuntos de testes especializados para as partes disponíveis, isto acrescenta custos adicionais ao sistema.

Um inspeção pode considerar atributos de qualidade mais amplos de um programa como conformidade com padrões, portabilidade e facilidade de manutenção, podendo procurar ineficiências, algoritmos inapropriados e estilo de programação que poderiam tornar difíceis a manutenção e atualização de sistema.

Inspeções levam tempo para serem organizadas e parecem diminuir a velocidade do processo de desenvolvimento. É difícil convencer um gerente extremamente pressionado de que esse tempo pode ser recuperado mais tarde pelo fato de que menos tempo será gasto no *debugging* do programa.

Concluindo, alguns estudos mostram que as inspeções são mais eficientes para descobrir defeitos do que teste de programas, sendo que 60% de erros de um programa podem ser detectados usando-se inspeções informais e as inspeções baseadas em argumentos de correção pode detectar mais de 90% dos erros de um programa. Vale ressaltar que as inspeções que incorrem no início os custos V&V do *software* e resultam em economias de custo somente depois que as equipes de desenvolvimento se tornam experientes em seu uso.

Na seção seguinte abordaremos o processo V&V.

8.4.1 Processo

O objetivo especifico das inspeções é encontrar os defeitos de programa, mais do que considerar questões mais amplas de projeto. Os defeitos podem ser erros de lógica, anomalias no código que possam indicar uma condição errônea ou não conformidade aos padrões do projeto ou organizacionais.

Este processo é realizado por uma equipe de pelo menos quatro pessoas.

A inspeção propriamente dita deve ser bem curta e deve enfocar a detecção de defeitos, conformidade aos padrões e programação de baixa qualidade. A equipe de inspeção não deve sugerir como esses defeitos devem ser corrigidos nem recomendar mudanças em outros componentes.

Antes de iniciar o processo de inspeção é essencial que:

Tenha uma especificação precisa do código a ser inspecionado. Sendo impossível inspecionar um componente no nível de detalhes necessário para detectar defeitos sem uma especificação completa.

Os membros da equipe devem estar familiarizados com os padrões organizacionais.

Uma versão atualizada e compilável do código tenha sido distribuída para todos os membros da equipe.

Veja abaixo os papéis no processo de inspeção:

Papel	**Descrição**
Autor e proprietário	Programador ou projetista responsável por produzir o programa ou o documento e pela correção de defeitos descobertos durante o processo de inspeção.
Inspetor	Encontra erros, omissões e inconsistências nos programas e documentos, identifica questões mais amplas fora do escopo da equipe de inspeção.
Leitor	Apresenta o código ou documento em uma reunião de inspeção.
Relator	Registra os resultados da reunião de inspeção.
Presidente ou moderador	Gerencia o processo e facilita a inspeção. Relata os resultados do processo ao moderador-chefe. Responsável pelo planejamento da inspeção.
Moderador-chefe	Responsável pelos aprimoramentos do processo de inspeção, pela atualização da

	lista de verificação, pelo desenvolvimento de padrões, etc.

Tabela 8 Papéis no processo de inspeção – tabela elaborada conforme tabela 22.1, capitulo 22 Verificação e Validação do Livro Engenharia de Software – Iam Sommerville

O processo de inspeção não deve durar mais que duas horas, focando a detecção de defeitos, conformidade aos padrões e programação de baixa qualidade. A equipe de inspeção não deve sugerir como esses defeitos devem ser corrigidos nem recomendar mudanças em outros componentes.

O autor faz as mudanças para corrigir os problemas identificados, em seguida o moderador decide se uma nova inspeção do código é necessária, decidindo que uma nova inspeção completa não é necessária e que os defeitos foram corrigidos com sucesso, aprovando o programa para ser liberado.

Cada organização deve desenvolver seu próprio checklist de inspeção baseado nos padrões locais e na prática, devendo ser atualizados regularmente conforme novos tipos de defeitos são encontrados.

Com quatro pessoas envolvidas em uma pesquisa de inspeção, o custo de 100 linhas de códigos é equivalente ao esforço de uma pessoa-dia.

Por fim concluísse que inspeções são mais eficientes para encontrar erros, e que os custos de teste de componente não são justificados. Inspeções de componentes combinadas com teste de sistema era a estratégia V&V de custo mais adequado.

Veja abaixo tabela de classe de defeito:

Classe de defeito	Verificação de Inspeção	Sim/Não	Obs.
Defeitos de Dados	Todas as variáveis de programa são iniciadas antes que seus valores sejam usados?	( ) sim ( ) não	
	O limite superior de vetores deve ser igual ao tamanho do vetor ou tamanho – 1?	( ) sim ( ) não	
	Se são usados strings de caracteres, um delimitador é explicitamente atribuído?	( ) sim ( ) não	
	Existe alguma possibilidade de *overflow* de *buffer*?	( ) sim ( ) não	
Defeitos de controle	Para cada declaração	( ) sim	

	condição, a condição está correta?	( ) não	
	Cada loop está terminando corretamente?	( ) sim ( ) não	
	As declarações compostas estão corretamente delimitadas entre parênteses?	( ) sim ( ) não	
	Em declarações '*case*', todos os casos possíveis são levados em conta?	( ) sim ( ) não	
	Se um comando '*break*' é necessário após cada caso nas declarações '*case*', ele foi incluído?	( ) sim ( ) não	
Defeitos de Entrada/Saída	Todas as variáveis de entradas são usadas?	( ) sim ( ) não	
	Todas as variáveis de saída têm valor atribuído antes de sua saída?	( ) sim ( ) não	
	Entradas inesperadas podem fazer com que os dados são corrompidos?	( ) sim ( ) não	
Defeitos de Interface	Todas as chamadas de funções e de métodos têm o número correto de parâmetros?	( ) sim ( ) não	
	Tipos de parâmetros reais e formais se combinam?	( ) sim ( ) não	
	Os parâmetros estão em ordem correta?	( ) sim ( ) não	
	Se os componentes acessam memória compartilhada, eles têm o mesmo modelo de estrutura de memória compartilhada?	( ) sim ( ) não	
Defeitos de gerenciamento/armazenamento	Se uma estrutura ligada é modificada, todas as ligações foram corretamente reatribuidas?	( ) sim ( ) não	
	Se o armazenamento dinâmico foi usado, o espaço foi corretamente alocado?	( ) sim ( ) não	
	O espaço de memória é liberado depois de não ser mais necessário?	( ) sim ( ) não	

Defeitos de gerenciamento de exceções	Todas as condições possível de erro foram consideradas?	( ) sim ( ) não	

Tabela 9 Classe de Defeito e sua verificação de inspeção - tabela elaborada conforme tabela 22.2, capitulo 22 Verificação e Validação do Livro Engenharia de Software – Iam Sommerville

A introdução de inspeções tem implicações para gerenciamento do projeto. A inspeção de programa é um processo público de detecção de erros comparado com o processo de teste mais privativo.

Concluísse que a medida que uma organização ganha experiência no processo de inspeção, ela pode usar os resultados de uma inspeção para ajudar no aprimoramento do processo.

A seguir falaremos da análise estática automatizada.

8.5 Análise estática automatizada

Aqui o programa é examinado sem ser executado. Geralmente dirigidas por *checklist* de erros e heurísticas que identificam erros comuns em diferentes linguagens de programação.

Analisadores Estáticos são ferramentas de *software* que varrem o texto-fonte de um programa e detectam possíveis defeitos e anomalias. Eles podem detectar se as declarações estão bem formuladas, fazer inferências sobre o fluxo de controle do programa e em muitos casos computam o conjunto de todos os valores possíveis para os dados de programa. Eles complementam os recursos de detecção de erros providos pelo compilador da linguagem.

As anomalias são freqüentemente um resultado de erros de programação ou omissões, de modo que eles enfatizam coisas que poderiam sair erradas quando o programa fosse executado, tais anomalias não são necessariamente defeitos.

Veja os estágios envolvidos na análise estática:

Análise do fluxo de controle – identifica e enfatiza os *loops* com vários pontos de saída e entrada e código inacessível que compreende um código cercado por declarações incondicionais ‘goto’ ou em uma ramificação de uma declaração condicional na qual a condição de guarda nunca pode ser verdadeira.

Análise de uso de dados – mostra como as variáveis são usadas, detecta variáveis sem prévia iniciação, variáveis escritas duas vezes sem uma tarefa de impedimento e variáveis declaradas, mas que nunca foram usadas. Este tipo de análise também descobre testes não eficientes nos quais as condições do teste é reduntante.

Análise de interface – verifica a consistência das declarações de rotina e de procedimento e seus usos. Na linguagem Java o compilador realiza tais verificações. Pode também detectar funções e procedimentos declarados e nunca usados ou resultado de funções que nunca são usados.

Análise de fluxo de informações – identifica as dependências entre variáveis de entrada e de saída, mostra como o valor de cada variável de programa é derivada de outros valores de variáveis. Podendo mostrar também as condições que afetam um valor de variável.

Análise de caminho – identifica todos os caminhos possíveis por meio do programa e estabelece as declarações executadas naquele caminho.

Classe de defeito	**Verificação de análise estática**	**Sim/Não**	**Obs**
Defeitos de Dados	Variáveis usadas antes da inicialização.	( ) sim ( ) não	
	Variáveis declaradas, mas nunca usadas.	( ) sim ( ) não	
	Variáveis atribuídas duas vezes, mas nunca usadas entre atribuições.	( ) sim ( ) não	
	Possíveis violações de limites de vetor.	( ) sim ( ) não	
	Variáveis não declaradas.	( ) sim ( ) não	
Defeitos de Controle	Código Inacessível	( ) sim ( ) não	
	Ramificações incondicionais em loops.	( ) sim ( ) não	
Defeitos de entrada/saida	Variáveis geradas duas vezes sem tarefa de impedimento.	( ) sim ( ) não	
Defeitos de Interface	Tipo de parâmetro que não combina.	( ) sim ( ) não	
	Número de parâmetro que não combina.	( ) sim ( ) não	
	Resultados de funções não usadas.	( ) sim ( ) não	
	Funções e procedimentos não chamados.	( ) sim ( ) não	

Defeitos de gerenciamento de armazenamento	Ponteiros não atribuídos.	( ) sim ( ) não	
	Aritmética de ponteiros.	( ) sim ( ) não	

Tabela 10 Verificação da análise estática automatizada - tabela elaborada conforme tabela 22.3, capitulo 22 Verificação e Validação do Livro Engenharia de Software – Iam Sommerville

Concluísse que os analisadores estáticos são viáveis em linguagem de programação que não possuem regras de tipagem estritas e a verificação de um compilador, um exemplo é a linguagem C. Assim tal análise pode descobrir um grande número de erros potenciais e reduzir significativamente os custos de teste. A abordagem de prevenção de erros em vez de detecção de erros é mais eficiente no aprimoramento da confiabilidade de programa.

8.6 Métodos Formais

São baseados em representações matemáticas de *software* usualmente como uma especificação formal. Estão relacionados a uma análise matemática da especificação.

Este método exige análises muito detalhadas de especificação de um sistema e de um programa e seu uso freqüentemente consome tempo sendo oneroso, assim fica restrito aos processos de desenvolvimento de software com segurança e proteção crítica.

8.6.1 Estágios

Uma especificação formal de sistema pode ser desenvolvida e analisada matematicamente para verificar inconsistências. Essa técnica é eficiente para descobrir erros de especificação e omissões.

A verificação ocorre por meio de argumentos matemáticos, se o código de um sistema de *software* está consistente com sua especificação. Eficiente na descoberta de erros de programação e de projeto.

Existe uma argumentação contra o uso deste método devido a exigência de notações especializadas usadas por pessoas especializadas não sendo compreendidas por especialistas de domínio de problema.

A verificação em um *software* não trivial requer grande quantidade de tempo e ferramentas especializadas, como provadores de teorema e habilidades matemáticas.

É possível afirmar que tal método conduz a sistemas mais confiáveis e seguros. Sendo uma especificação formal menos propensa a conter anomalias que devem ser resolvidas pelo projetista do sistema. Dessa forma tal especificação não garante que o *software* será confiável no uso prático, tendo como razões:

A especificação pode não refletir os requisitos reais dos usuários do sistema.

As provas de programa são amplas e complexas, portanto podem conter erros.

E por fim a prova de assumir um padrão de uso incorreto. Se o sistema não é usado conforme o previsto, a prova pode ser inválida.

Este mesmo nível de confiabilidade oferecido por este método pode ser alcançado de maneira dispendiosa com o uso de outras técnicas como: inspeções e testes de sistemas.

O processo *cleanroom* pode ser usado para apoiar o processo de verificação formal. Tal processo adota uma filosofia de desenvolvimento de *software* que usa métodos formais para apoiar inspeção rigorosa de *software*. Tem como objetivo gerar *software* com defeito zero. Vêem substituir o teste unitário dos componentes de sistema por meio de inspeções para verificar a consistência desses componentes com suas especificações. O *Cleanroom* funciona quando praticado por engenheiros habilidosos e comprometidos, não se sabe de seu uso em organizações menos comprometida. Esta abordagem ou qualquer outra que utiliza métodos formais são usados para organizações tecnicamente menos avançadas ainda representa um desafio.

Por fim concluísse que o uso dessa abordagem está aumentando de acordo com a demanda dos clientes e a medida que engenheiros familiarizam-se com essas técnicas.

Na seção a seguir falaremos de gerenciamento de defeitos.

9 GERENCIAMENTO DE DEFEITOS

Será abordada nesta seção a gestão de defeitos fazendo uma correlação com a fase de teste.

O principal objetivo deste processo é evitar os defeitos.

O processo de gestão é um processo de melhoria continua, onde temos as fases de prevenir defeitos, gerenciar *baselines*, identificar defeitos e solucionar defeitos. Tudo isto pode ser visto nos relatórios de gestão.

Assim devemos identificar os erros nos requisitos.

Veja com mais detalhes as fases do processo de gestão de defeitos:

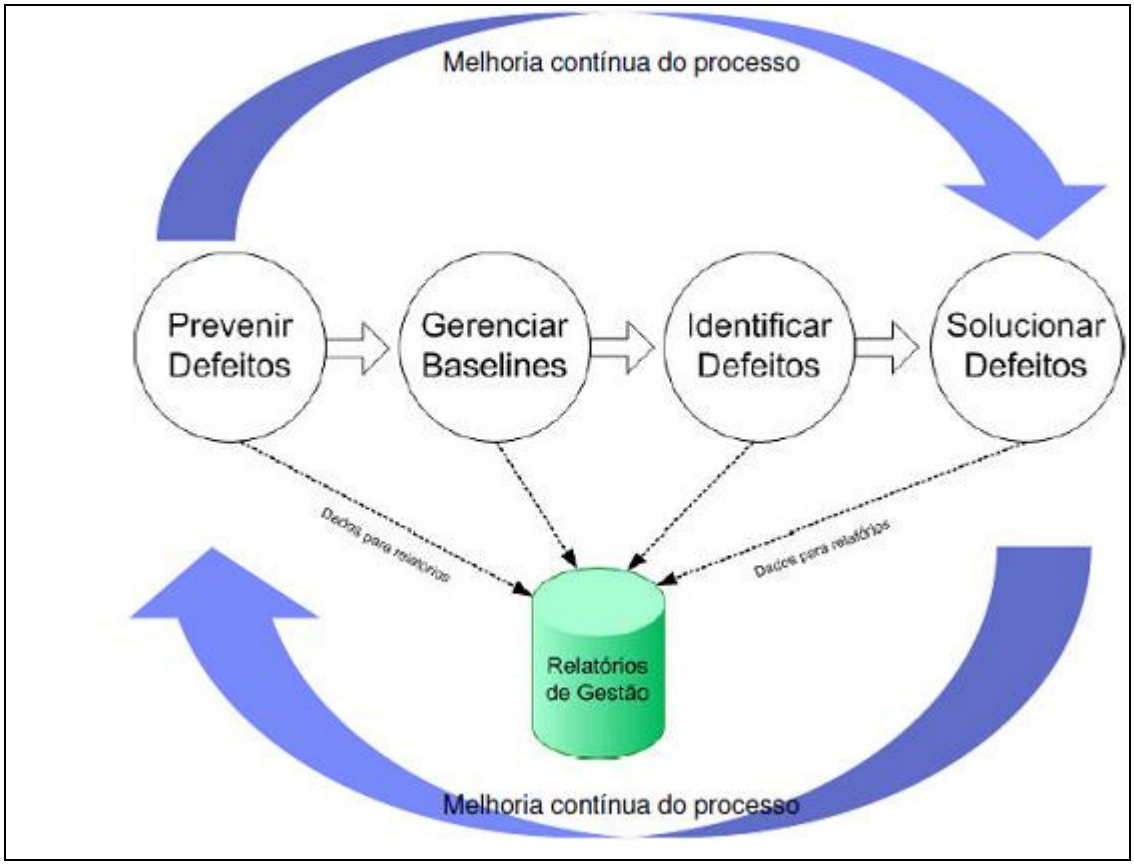


Figura 3 Processo de Gestão (Kelvin Weiss, 14/03/10, página 3 - Figura retirada do curso Fundamentos em Teste de Software, módulo 8)

9.1 Prevenir defeitos

A prevenção dos defeitos se dá pelo uso de técnicas e processos que ajudam a detectar e evitar erros antes que estes se propaguem para outras fases do desenvolvimento. Para isto o processo de teste deve ser iniciado juntamente com o desenvolvimento. O uso das técnicas de teste estático (teste de verificação) é fundamental para esta atividade.

Suas características são:

É uma das fases mais importantes do ciclo de vida do desenvolvimento de *software*;

Com impacto direto no controle dos custos e na qualidade da entrega;

O custo da prevenção é muito menor do que o custo do re-trabalho;

É mais eficaz durante a fase de Requisitos, pois é a fase que se encontra erros de escopo.

A melhor prevenção é a execução correta das atividades, esta exige:

Adequação do processo: ao negócio, à organização e à aplicação;

Emprego de tecnologia adequada;

E principalmente capacitação das pessoas.

Algumas medidas podem ser tomadas para minimizar os riscos do projeto e conseqüentemente os riscos de *software* ir para produção com defeitos. Para isto é necessário as seguintes ações:

Treinamento e Educação;

Metodologia e Padrões;

Desenho Defensivo;

Código defensivo.

Tudo isto pode parecer exaustivo e impossível de manter, mas o retorno é disponibilizar aos clientes aplicativos confiáveis e de qualidade.

9.2 Gerenciar *Baseline*

Baseline é uma "foto" de um artefato ou sistema no repositório do projeto. Em suma é criar uma versão de documento congelada não sendo mais alteradas, tais alterações devem ser feitas em uma nova versão do arquivo.

Um produto a ser entregue pode ser considerado uma *baseline* quando atinge o marco pré-definido (ponto de corte) do seu processo de desenvolvimento que deve ser enviado para a equipe de teste que cria uma *baseline*.

O processo de *baseline* acontece da seguinte forma:

Identificar produtos chaves

Os produtos chaves de uma *baseline* são: casos de teste, planos de teste.

Criar *baselines*.

Gerenciar *baselines.*

9.3 Identificação do Defeito

O defeito é considerado identificado quando formalmente for reconhecido pela equipe de desenvolvimento, pois nem tudo reportado a equipe é um defeito podendo ser ocorrências.

Na identificação de defeito temos algumas fases como:

Identificação de Defeito – aqui utilizamos técnicas estáticas (execução conceitual do programa), dinâmicas (o programa deve ser executado) e operacionais (encontra defeitos por acaso durante a utilização do *software*.

Reportar defeitos - utilizar ferramenta de Gestão de Defeitos seja ela manual ou automática. A seguir são citadas algumas técnicas que devem ser observadas para reportar defeitos:

Resumir – Reportar claramente, mas de forma resumida;

Precisão – Isto é um defeito ou poderia ser um erro do testador/usuário, erro de entendimento;

Neutralizar – Apenas fatos;

Generalizar – Procurar entender o problema de forma genérica;

Reproduzir – Reproduza um defeito ao menos duas vezes antes de reportá-lo;

Impacto – Qual o impacto deste defeito para o cliente?

Evidência – Evidencie a existência do defeito encontrado, ou seja, registre as telas de defeito dando um *Print Scrn*.

Reconhecer defeitos – decidir se o defeito é válido ou não.

Após o recebimento do defeito o desenvolvedor deve reconhecer o defeito como válido ou não. É aconselhável registrar as evidências do defeito (casos de teste) e enviar para o desenvolvedor juntamente com o caso de teste executado. Isso agilizará o processo de reconhecimento.

O desenvolvedor sempre consultará a documentação do sistema para reconhecer o defeito. Por via de regra, só considerará defeito se o sistema tiver um comportamento diferente do escrito nos requisitos. Sempre que houver divergências se um defeito é

válido ou não, deve-se escalar a decisão para a gerência e então ela definirá se o defeito é válido ou não.

9.4 Solução do defeito

Sendo responsabilidade da equipe de desenvolvimento. A solução do defeito acontece com as seguintes fases:

Priorizar correção – a priorização deve acontecer com a seguinte classificação:

1 – Alta

2 – Média

3 – Baixa

Programar correção - a correção será de acordo com a prioridade definida.

Corrigir defeito - executar a correção e revisar testware.

Reportar solução - notificar a correção, esclarecendo a natureza desta correção e definir quando e como esta será liberada.

A melhoria do processo procura revisar o processo, identificar as causas dos defeitos, tomar ações para que os defeitos não ocorram novamente e por último o TMM (*Test Maturity Model*) provê suporte às organizações na melhoria do Processo de Testes. Podemos ter no CMMI (*Capability Maturity Model Integration*) um framework para guiar a melhoria no processo de desenvolvimento e teste.

9.5 Relatórios de Gestão

Todos os defeitos encontrados devem ser registrados nos **Relatórios de Gestão.**

Os defeitos são registrados e armazenados com os seguintes objetivos:

Corrigir defeitos encontrados;

Ter visão da situação do *software*;

Prover relatórios de gestão;

Prover dados estatísticos para melhoria do processo de desenvolvimento e testes de *software*.

9.6 Taxonomia de Defeitos

Taxonomia é a maneira que classificamos os defeitos.

Quanto melhor for a Taxonomia dos Defeitos, melhor será a Gestão dos Defeitos, pois as métricas coletadas terão informações mais precisas sobre a origem do problema, ocasionando numa eficaz melhoria do processo. Segue abaixo uma visão macro dessa ciência para a gerência de defeitos.

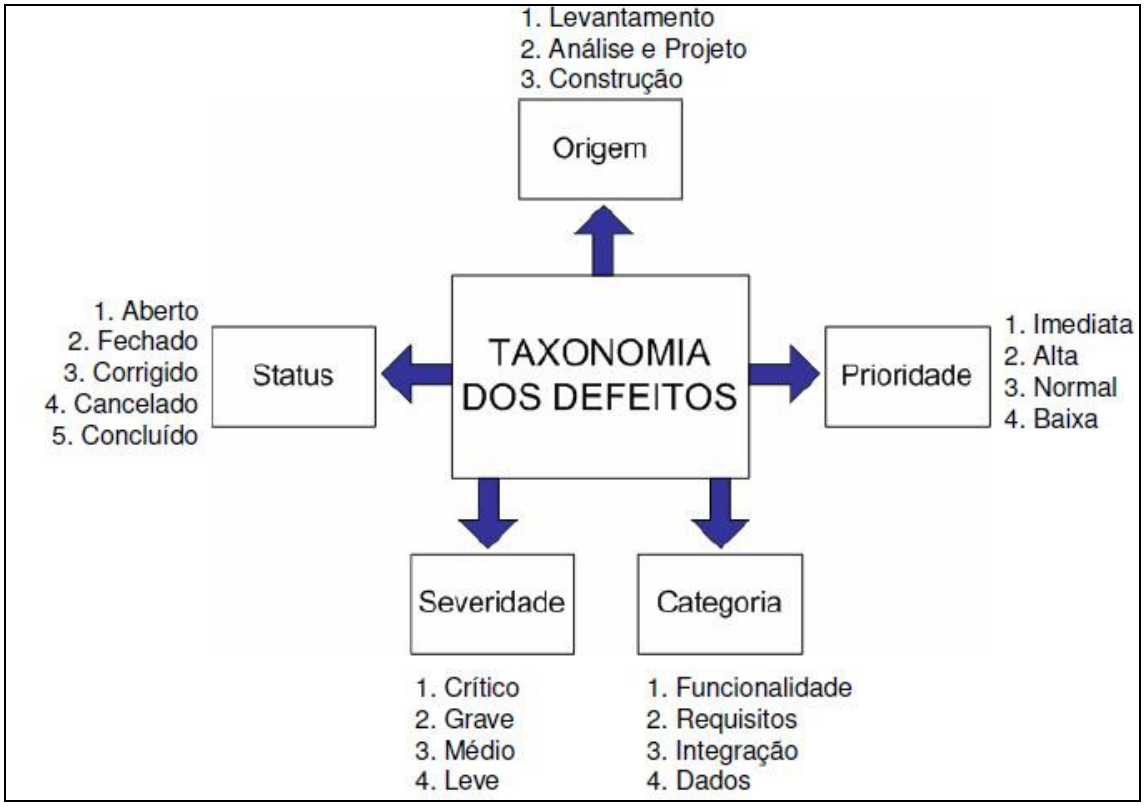


Figura 4 Taxonomia dos defeitos (Kelvin Weiss, 15/04/10, página 17 - Figura retirada do curso Fundamentos em Teste de Software, módulo 8)

Origem – classifica a origem do desenvolvimento.

Prioridade - classifica a prioridade de correção pela equipe de desenvolvimento.

Categoria – dá a visão dos tipos mais freqüentes de defeitos.

Severidade – indica o impacto que o defeito pode causar ao negócio ou ao *software*.

Status – apresenta o status do defeito.

A seguir apresentaremos uma lista de ferramentas de gestão de defeitos.

9.7 Ferramentas de Gestão de Defeitos

As ferramentas de gestão de defeitos permitem que seja realizado o rastreamento e correção dos defeitos. Todos os defeitos encontrados no *software* devem ser registrados identificando a funcionalidade onde ocorreu o defeito.

Veja alguns exemplos destas ferramentas:

Bugzilla

Mantis

Rational *ClearQuest*

TestDirector (Mercury)

TrackRecord (*Compuware*)

Jira

O gerenciamento de defeitos tem como intuito evitar defeitos. É um processo orientado para minimizar os riscos. Sendo uma gestão integrada ao processo de desenvolvimento. Sua automatização é recomendada sempre que possível.

As informações coletadas devem ser usadas para melhoria de processo.

Vale lembrar que cada empresa deve adaptar o processo de teste a sua realidade.

10 RELATÓRIOS DE TESTES

Aqui serão apresentados os objetivos dos relatórios de testes, bem como apresentar modelos do mesmo.

O histórico dos resultados reais do teste, problemas ou peculiaridades é registrado aqui, podendo ser anexada a Especificação de Testes se desejarem. As informações contidas nestes relatórios podem ser vitais durante a manutenção do *software*. Referências e apêndices adequados são também apresentados.

Os relatórios de testes são elaborados para documentar os resultados dos testes, como definidos no Plano de Testes.

Eles visam cumprir três objetivos:

_ Definir o escopo dos testes;

_ Apresentar os resultados dos testes;

_ Apresentar conclusões e recomendações sobre os resultados.

A partir desses relatórios teremos condições de comparar o que planejamos com o que executamos.

10.1 Propósitos dos relatórios

Os propósitos dos relatórios de testes de imediato é prover informações sobre o sistema, que permita determinar se este está pronto para entrar em produção, bem como avaliar potenciais conseqüências e iniciar ações apropriadas para minimizá-las.

Em longo prazo prove rastrear problemas eventuais em produção, analisar o processo de trabalho e evidenciar o teste.

10.2 Categorias

Existem três categorias:

A) Relatórios da Situação do Projeto - Os relatórios da situação do projeto são extraídos para equipe do projeto e gerência Sênior. Informa os resumos do projeto, podendo ser diários, semanais ou mensais, podendo mostrar as estatística em forma de gráfico.

B) Relatórios intermediários de Teste - São aqueles que fornecem informações sobre a execução dos testes. Um exemplo é o relatório de ocorrências.

C) Relatórios finais dos Testes - São definidos para apoiar as decisões relativas à implantação do *software*. Eles devem indicar se o *software* está completo e correto. Deve ser elaborado após a execução de cada teste.

A norma IEEE 829 prevê os seguintes relatórios: de log, de ocorrência e de teste.

10. 3 Diretrizes para relatórios

Existem algumas diretrizes que devem ser seguidas para preparação e utilização das informações dos relatórios, são elas:

Desenvolver uma linha base (*baseline*) dados que serviram de base para o relatório;

Usar modelos e *checklists* de boas práticas;

Permitir que a equipe do projeto revise e comente o rascunho antes de sua finalização;

Não incluir nomes ou estabelecer culpados e enfatizar a qualidade;

Limitar os relatórios aos itens importantes;

Eliminar detalhes e pequenos problemas;

Entregar os relatórios ao Gerente do Projeto;

Envolver as equipes de Teste e de Projeto no apontamento e recomendações aos itens.

Visto que a elaboração e interpretação de relatórios envolve comunicação entre as partes interessadas, segue abaixo diretrizes para gerenciar tal comunicação.

10.4 Gerência de Comunicação segundo o PMBOK

A gerência de comunicação de um projeto deve incluir os processos necessários para a geração, coleta, distribuição, armazenamento e descarte das informações.

O gerenciamento da comunicação deve estabelecer quais as informações que serão distribuídas, quem receberá as informações e em que momento.

Quatro etapas compõem a gerencia de comunicação segundo o PMBOK:

Etapas	Entrada	Saída
Planejamento das comunicações	_ Requisitos de Comunicação (evita informações desnecessárias, identificando quais será necessárias ao projeto); _ Tecnologia de Comunicações (e-mail); _ Restrições.	_ Plano de gerência das comunicações
Distribuição das informações	_ Resultados do Trabalho; _ Plano de gerência das Comunicações (quem receberá as informações, como estas serão enviadas, tipo e periodicidade de	_ Registros do projeto; _ Relatórios formais do Projeto; _ Apresentações do Projeto;

	envio); _ Plano de Testes.	
Relatórios de Desempenho Este tipo de relatório permite ao gerente do projeto tomar decisões.	_ Plano de Testes; _ Resultado do Trabalho; _ Registro do projeto.	_ Relatório de progresso; _ Relatórios de situação; _ Relatório Log de Teste; _ Relatório de Ocorrências.
Encerramento	_ Documentos; _ Registros; _ E outras informações do Projeto.	_ Plano de Gerência de Comunicações; _ Relatório de Sumário de Teste (é um relatório básico do encerramento dos testes. Deve ser enviado para todos os envolvidos de acordo com o planejamento das comunicações).

Tabela 11 Etapas compõem a gerencia de comunicação segundo o PMBOK

11 QUALIDADE DE *SOFTWARE*

Nesta seção apresentamos um resumo do conceito de teste e a sua relação com a qualidade do produto de *software*.

O teste de *software* é uma das fases do processo de Engenharia de *Software* que visa atingir um nível de qualidade de produto superior. O objetivo, por paradoxal que pareça, é mesmo o de encontrar defeitos no produto, para que estes possam ser corrigidos pela equipe de analistas/desenvolvedores, antes da entrega final do produto. A maioria das pessoas pensa que o teste de *software* serve para demonstrar o correto funcionamento de um programa, quando na verdade ele é utilizado como um processo da engenharia de *software* para encontrar defeitos. De uma forma simples, testar um *software* significa verificar através de uma execução controlada se o seu comportamento ocorre de acordo com o especificado. O objetivo principal desta tarefa é encontrar o número máximo de erros dispondo do mínimo de esforço, ou seja, mostrar aos que desenvolvem se os resultados estão ou não de acordo com os padrões estabelecidos. A qualidade de *software* está relacionada com um desenvolvimento organizado de *software* tendo como premissa uma metodologia de trabalho. Tendo como base conceitos que visem à construção de um produto de *software* de qualidade. Dentro desta

metodologia estão definidos os passos necessários para chegar ao produto final esperado e a fase de teste compõe um desses passos.

É importante observar que a qualidade em *software* não se resume a testar sistemas e suas aplicações. Há mais coisas envolvidas como; métricas, metodologias (também aplicadas ao desenvolvimento), gerenciamento de projetos e ferramentas exclusivas de testes para automação, testes unitários entre outros.

A obtenção da qualidade está associada à execução e implementação de diferentes tipos de teste. Cada tipo de teste tem objetivo e técnica de suporte específico. Cada técnica de teste foca em testar uma ou mais características ou atributos do objetivo do teste.

A seguir falaremos um pouco sobre as características da qualidade:

11 Características da qualidade

A Norma ISO 9126-1 define seis características de qualidade que o software deve atender, a saber:

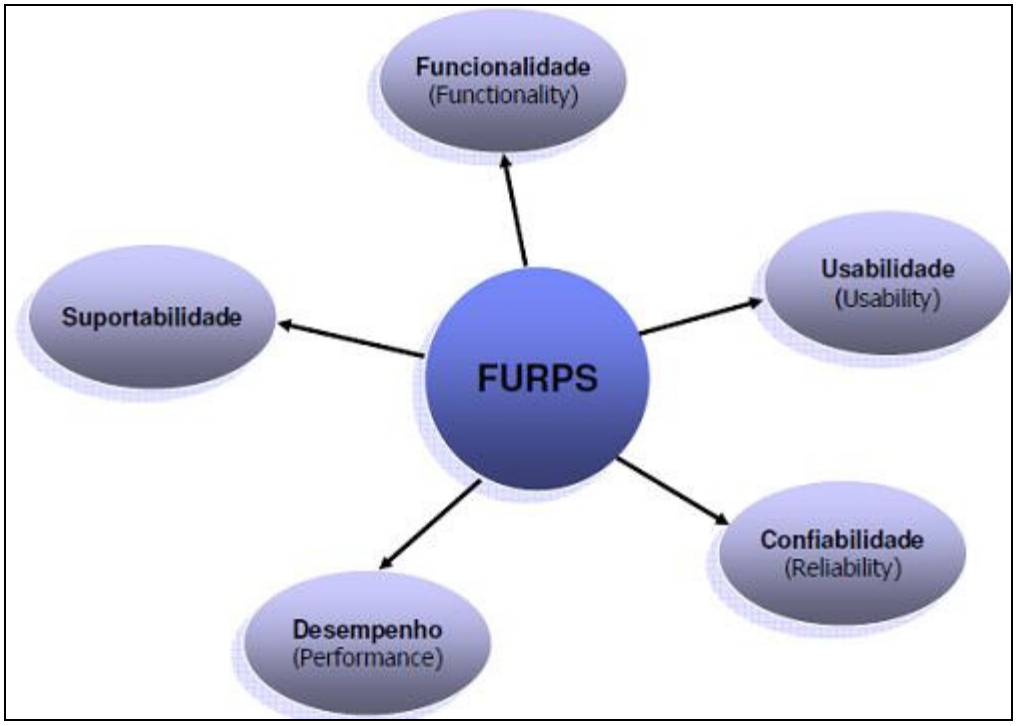


Figura 5 FURPS (Kelvin Weiss, 15/04/10, página 17 - Figura retirada do curso Fundamentos em Teste de Software, módulo 2)

Funcionalidade – especifica as ações que um *software* é capaz de realizar sem levar obstáculos físicos em consideração. Verifica a capacidade do sistema em prover funcionalidades definidas que atendam as necessidades do usuário, quando usado sob determinadas condições preestabelecidas. Conformidade dos requisitos estabelecidos.

Usabilidade - avalia fatores humanos, estética da aplicação, consistência da interface, ajuda da aplicação, agentes e documentação de usuário e matérias de treinamento. Observados os aspectos que facilitam a utilização do sistema. Capacidade do *software* em ser entendido, aprendido e utilizado sob condições estabelecidas de utilização.

Confiabilidade – previsão de falhas, segurança de acesso, segurança de ambiente, capacidade de recuperação de carga. Avaliação o quão o *software* é capaz de manter-se instável.

Desempenho – avalia questões de velocidade, disponibilidade, tempo de respostas, uso de recursos, etc. O produto de software é capaz de manter seu nível de desempenho, ao longo do tempo, nas condições estabelecidas de utilização.

Suportabilidade/Portabilidade – avalia a capacidade de teste, a adaptabilidade, flexibilidade e a compatibilidade de versões. Facilidade de o *software* poder ser transferido de um ambiente para outro.

_ **Manutenibilidade**: refere-se ao esforço necessário para a realização de alterações específicas no produto de *software*. Verifica o esforço necessário para realizar uma manutenção.

Todas estas características poderão ser comprovadas mediante as estratégia e técnicas de teste.

Mediante ao que foi apresentado no decorrer desta pesquisa podemos concluir que se aliando a um bom processo de "Testes de *Softwares*" é possível o alcance da desejada "Qualidade de *Software*".

12 ESTUDO DE CASO

Diante das estratégias de teste existentes e abordadas na parte teórica desta pesquisa, foi feito um estudo de caso na empresa Varuna Tecnologia Ltda, uma *software house* que desenvolve sistemas nas seguintes áreas: Engenharia Civil e Pesada, Comércio Varejista, Corretora de Valores e Prestadores de Serviços para emissão de NFS-e.

O Produto a ser Testado: Help Desk (HD) – que é uma ferramenta gerencial para controle dos chamados e execução de suas respectivas tarefas de nossos clientes (chamadas externas) e também de nossos colaboradores (chamadas internas), bem como documentação dos Relatórios de Visitas Técnicas (RVT's).

O estudo de caso compreende:

a) Escolher uma das estratégias estudadas na parte teórica desta pesquisa;

b) Escolher as técnicas de teste de acordo com o estágio do projeto;

c) Escolher um dos padrões existentes no mercado para documentar as fases de teste.

a) Neste item, adotamos a estratégia de Teste de Sistema. Os motivos que nos levou a está escolha foram:

-É tradicionalmente uma estratégia adotada quando o software está funcionando como um todo;

-Seu objetivo é a funcionalidade do sistema final, cuja finalidade principal é exercitar por completo o sistema baseado em computador;

-É uma série de teste de integração e validação.

b) Dentro dos tipos de teste relacionados a teste de sistemas foram adotados os seguintes testes: Funcionalidade, Interface do Usuário e Segurança, estes serão apresentados no Plano de Teste Mestre (PTM), conforme mostra a tabela 6 do anexo 1 e abordados na Especificação e Procedimentos de Casos de Teste (EPCT) anexo 2.

Todas as técnicas utilizadas são técnicas de teste de Caixa-Preta.

c) Para o estudo de caso em questão adotamos e adaptamos *templates* elaborados mediante os templates da norma IEEE 829 adequando ao perfil da empresa.

A elaboração dos cenários e dos casos de testes facilita a aplicação dos testes, bem como servirá de documentação, independente de poder validar as especificações de requisitos caso as tenha ou não.

O Plano de Teste Mestre (PTM) irá apresentar a importância do planejamento para elaboração de casos de teste, abordando as estratégias de testes bem como aplicar alguma das técnicas discutidas no item: Técnicas de Teste no capítulo 7. O relatório de Especificação e Procedimentos de Casos de Teste (EPCT) contém os cenários e casos de teste a serem aplicados no projeto em questão. E por fim o Relatório de Log de Teste (LDT) apresentará uma síntese do que foi obtido na realização dos casos de teste.

Veja na seção de anexos os templates utilizados:

Anexo 1 – 1 PTM_Varuna_HD_V.1.0,

Anexo 2 – 3 EPCT_Varuna_HD_V.1.0,

Anexo 3 – 5 LDT_Varuna_HD_V.1.0.

12.1 Dificuldades da aplicação

No decorrer da execução do estudo de caso, deparamos com as seguintes dificuldades:
Resistência à elaboração de documentos por parte da cultura de profissões técnicas;
O pouco espaço de tempo dificulta o planejamento e aplicação dos testes;
Nem sempre a informação necessária ao planejamento e projeto do teste está facilmente disponível;
Desconhecimento de padrões de documentação pelos testadores;
Normalmente, as ferramentas que auxiliam o processo de documentação do teste são caras e complexas;
Os padrões de documentação existentes não são adotados;
Falta de consciência da importância da documentação.

12.2 Melhorias observadas

Após concluir o estudo de caso foram observadas algumas melhorias:

Menor número de defeitos descobertos após a liberação do software;
Mudança de atitude da equipe de programação: maior atenção às tarefas de verificação;
Melhoria no processo de desenvolvimento de software;
Mudança de atitude dos clientes: maior tolerância quanto aos prazos de liberação dos produtos.

12.2.1 Em se tratando da documentação

Facilidade para controlar atividades de testes;

Reduz a duplicação de esforço depois de ter uma base de documentação já criada (Permite o reaproveitamento de esforços passados);
Fornece aos clientes evidência da qualidade do software;
Melhora na qualidade das atividades de teste.

Considerando a fase de testes e sua importância no processo de desenvolvimento de *software*, talvez com um estudo mais crítico e aprofundado seja possível a adoção de alguns artefatos na empresa a qual trabalho.

13 CONCLUSÃO

Abordando a fase de Testes, como um todo será apresentado alguns dos benefícios, restrições e dificuldades.

Em se tratando de benefícios, com a adoção das estratégias e técnicas, é possível obter:

- Qualidade de *software*;
- Aplicativos finais sendo executados com menor número de erros possíveis;
- Facilidade na operação de manutenção de *software*;
- Controle sobre desenvolvimento dentro de custos, prazos e níveis de qualidade desejados;

Apesar dos benefícios, é preciso acordar que os mesmos não virão de maneira imediata. É necessário adquirir treinamento adequado, adaptação da metodologia no contexto ao qual ela será utilizada, apoio especializado para as equipes de desenvolvimento e de testes e tempo para a absorção da mesma.

Em relação às restrições e dificuldades:

Mediante estas requisições algumas empresas têm restrições na adoção de tal metodologia. Como foi mencionado nos benefícios, umas das dificuldades muitas das vezes encontradas em empresas de pequeno porte é ter pessoal preparado ou até em conseguir mão de obra especializada nesta área, podendo assim torna-se um processo inviável financeiramente. Além da falta de profissionais especializados na área podemos citar:

Desconhecimento das técnicas de teste;

Poucos livros tratam o teste na prática;

Desconhecimento da relação custo benefício do teste;

O teste só é lembrado depois do software pronto;

Cursos de Engenharia de Software nas Universidades não tratam suficientemente as técnicas de teste;

Geralmente, a atividade de teste é executada sem um método que direcione o esforço para maximizar a descoberta de defeitos;

O teste é visto numa abordagem pessimista, como um conjunto de tarefas não produtivas.

Assim se conclui que para empresas de pequeno porte é necessário rever algumas atividades e adaptá-las a estrutura da empresa, bem como rever quais artefatos serão úteis e poderão ser adaptáveis.

REFERÊNCIAS

Curso e-learning, 2010, Divinópolis. **Fundamentos em Teste de *Software***. São Paulo: Editora TI Exames, 2010. 9 v.

MYERS, Glenford J. *The Art of Software Testing*. United States: Kindle Edition, 1979.

PRESSMAN, Roger S. *Engenharia de Software*. 6ª Ed. São Paulo: McGraw-Hill, 2006. Cap. 13 e 14.

SOMMERVILLE, Ian. *Engenharia de Software*. 8° Ed. São Paulo: Pearson, 2007. cap. 22,23,24 e 27.

TestExpert. **Templates da Norma IEEE 829 para DOWNLOAD**. Disponível em: < http://www.testexpert.com.br/?q=node/1666>. Acesso em: 15/07/10 as 22:35.

Wikipédia a enciclopédia livre. **Teste de Unidade**. Disponível em: <http://pt.wikipedia.org/wiki/Teste_de_unidade >. Acesso em: 25/04/10 as 22:52.

Wikipédia a enciclopédia livre. **Ian Sommerville.** Disponível em: < http://en.wikipedia.org/wiki/Ian_Sommerville_(academic) >. Acesso em: 15/07/10 as 22:16.

Wikipédia a enciclopédia livre. **Roger Pressman**. Disponível em: < http://pt.wikipedia.org/wiki/Roger_Pressman>. Acesso em: 15/07/10 as 22:27.

Varuna Tecnologia Ltda

PLANO DE TESTE

Projeto: Help Desk (HD)

Versão/Revisão 1.0.0.36_01

Responsáveis: Simone Moreira Cunha

Divinópolis – MG

Histórico de Revisão

Data	Versão	Comentário	Autor
11/06/2010	1.0	A nomenclatura para o nome desse documento seguirá o seguinte padrão: ordem_nome do artefato_nome do software_sistema_versão do documento.	Simone M. Cunha
16/06/10	1.1	Revisão	Simone M. Cunha

Conteúdo

Índice de Tabelas

1. Material de Referência

Versão/Revisão	Nome	Descrição

Tabela 1 Material de Referência

2. Introdução

Esse documento descreve o *plano de teste* do sistema **Help Desk (HD)**. Ele contém as estratégias de teste e o esforço necessário para sua realização. Na estratégia de teste estão definidos os tipos de teste que serão executados na iteração e os objetivos que devem ser atingidos.

Este documento atende os padrões da norma IEEE 829 constantes nos seguintes *templates*:

SQE Test Plan Template – Plano de Teste;
Test Design Specification Template – Especificação de Design de Teste no item 4;

2.1 Visão Geral do Documento

Nessa seção damos uma visão geral do documento, possibilitando um bom uso do mesmo. As seções a seguir fornecem informações sobre a execução dos testes do sistema **Help Desk**, e estão organizadas da seguinte forma:

- **Seção 3 – Glossário:** Incluir todos os termos críticos para ajudar a comunicação. Termos e Definições-chaves.
- **Seção 4 – Configuração de teste:** Identificar um conjunto de configuração a ser testada: software, estrutura de dados, recursos necessários.
- **Seção 4.1 – Itens que serão testados:** Descreve os itens que serão testados.
- **Seção 4.2 – Itens que não serão testados:** Descreve os itens que não serão testados.
- **Seção 5 – Questões de Riscos:** Identificar os pontos críticos para o sucesso do produto apontando as áreas críticas.
- **Seção 6 – Estratégia:** Descrevem os estágios e os tipos de teste contemplados no projeto, os testes que serão realizados por estágio e os critérios de conclusão dos mesmos. Bem como as técnicas utilizadas.
- **Seção 7 – Registro de Erros Encontrados:** Informa como se dará o registro dos erros encontrados durante o período de testes.

- **Seção 8 – Recursos:** Apresenta os recursos humanos, de *hardware* e de *software* necessários para a realização dos testes.

- **Seção 9 – Cronograma:** Apresenta o cronograma das atividades.

- **Seção 10 – Produtos Entregues:** Relação de todos os artefatos que serão gerados pelo processo de testes (Ex: Evidências de teste, casos de teste, relatório dos defeitos encontrados, relatório dos defeitos corrigidos, etc).

- **Seção 11 – Aprovações:** Descreve os aprovadores do plano de teste.

3. Glossário

Incluir todos os termos críticos para ajudar a comunicação.
Termos e Definições-chaves.

BVA	Análise de Valor Limite
CSU	Caso de Uso
CT	Caso de Teste
DFD	Diagrama de Fluxo de Dados
DS	Diagrama de Sequência
EPCT	Especificação e Procedimentos de Csso de Teste
HD	HelpDesk
IE	Internet Explore
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
RVT	Relatório de Visita Técnica
SQA	*Software Quality Assurance*
SQL	*Structured Query Language*
TA	Teste de Aceitação
TAA	Teste de Ambiente de Aceitação
TI	Teste de Integração
TS	Teste de Sistema
TU	Teste Unitário

4. Configuração de Teste

4.1 Itens que serão testados

Os itens a serem testados são os seguintes:

1 Login;
  1.1 Acesso;
2 Relatório de Visita Técnica

2.1 Dados do RVT;
2.1.1 Inserir;
2.1.2 Salvar;
2.2 Participante (s);
2.3 Consulta do Relatório de Visita Técnica;
3 Chamados
3.1 Cadastro;
3.1.1 Inserir;
3.1.2 Salvar;
3.2 Consulta de Chamados
3.2.1 Visualizar;
3.2.2 Liberação de Chamado;
3.2.4 Histórico.
4 Gerenciamento (Tarefas do Colaborador)
4.1 Tarefas Pendentes;
4.1.1 Iniciar Execução da Tarefa.
4.2 Tarefas Iniciadas;
4.2.1 Finalizar/Paralisar Tarefa.
4.2.2 Comentar tarefa.
4.3 Tarefas Paralisadas;
4.3.1 Tarefas pausadas;
4.3.2 Reiniciar Execução da Tarefa.
5 Cliente
5.1 Cadastro
5.1.1 Inserir;
5.1.3 Salvar;
5.2 Consulta Cliente
6 Publicação On Line de Relatórios
6.1 Cadastro
6.1.1 Inserir;
6.1.3 Salvar;
6.1.6 Relatório/Módulo;
6.2 Consulta
7 Atualização Remota de Banco de Dados
7.1 Cadastro
7.1.1 Inserir;
7.1.3 Salvar;
7.2 Consulta Atualização Remota de Banco de Dados;

4.2 Itens que não serão testados

1 Login
1.2 Configuração de Acesso;
2 Relatório de Visita Técnica
2.1 Dados do RVT
2.1.2 Editar;
2.1.4 Cancelar;
2.1.5 Excluir;
2.1.6 Imprimir.

2.2 Participante (s);
2.3.1 Visualizar;
2.3.2 Excluir Faturamento;
2.3.3 Faturar.
3 Chamados
3.1 Cadastro
3.1.2 Editar;
3.1.4 Cancelar;
3.1.5 Excluir.
3.2 Consulta
3.2.3 Validar Execução do Chamado;
3.2.5 Associar anexos ao chamado.
5 Cliente
5.1 Cadastro
5.1.2 Editar;
5.1.4 Cancelar;
5.1.5 Excluir.
5.3 Adições de Contatos;
5.4 Adições de Acesso Remoto.
6 Publicação On Line de Relatórios
6.1 Cadastro
6.1.1 Relatório/Módulo;
6.1.1.2 Editar;
6.1.1.4 Cancelar;
6.1.1.5 Excluir.
7 Atualização Remota de Banco de Dados
7.1 Cadastro
7.1.2 Editar;
7.1.4 Cancelar;
7.1.5 Excluir.

5. Questões de Riscos

Riscos de Planejamento e Contingências

Riscos	Hipóteses de Planejamento	Opções de Estratégias
Não conseguir aplicar todos os casos de teste elaborados no documento 3 ECT.	Seguir o cronograma apresentado no item 9.	Testar apenas os casos de teste mas significativos.

Tabela 2 Riscos de Planejamento e Contingências

Riscos Técnicos (do produto)

Funções Essenciais	Confiabilidade	Usabilidade	Segurança/Privacidade	Riscos/Probabilidade de falha
Controlar os RVT's através do sistema bem como promover				O sistema pode encontra indisponível no momento da documentação da

o faturamento das mesmas.				RVT (*in loco*).
Apresentar para o cliente o RVT referente a um determinado atendimento.				Falha na impressão do RVT.
Registrar os chamados, sejam internos ou externos.				O sistema pode encontra indisponível no momento da documentação do chamado.
Controle dos Chamados Liberados e não liberados, gerando uma notificação por e-mail ao colaborador.				Falha no envio de e-mail para notificação do colaborador.
Designação de tarefas aos colaborados para um determinado chamado, notificação por e-mail ao colaborador.				Falha no envio de e-mail para notificação do colaborador.
Gerenciamento de tarefas: Pendentes, Iniciadas e Paralisadas.				
Cadastro de clientes bem como controle de seus dados para acesso remoto.				
Publicação On Line de Relatórios no site de armazenamento dos respectivos relatórios.				Sistema de armazenamento pode está indisponível.
Atualização Remota de Banco de				Sistema de armazenamento pode está indisponível.

Dados no recipiente responsável.				

Tabela 3 Riscos Técnicos (do produto)

Obs.: Algumas dessas falhas podem ser provenientes a erros dos usuários ao manusear o sistema.

Riscos para o negócio do cliente

Riscos	Hipóteses de Planejamento	Opções de Estratégias

Tabela 4 Riscos para o negócio do cliente

Quando analisamos riscos em projetos de teste de software, geralmente os categorizamos em:

_ **Riscos de projeto**: são riscos ligados diretamente ao projeto. Fatores ligados: requisitos, pessoal, recursos, clientes e cronograma.
_ **Riscos técnicos**: são riscos relacionados à qualidade do software a ser desenvolvido.
_ **Riscos para o negócio do cliente**: são riscos relacionados a defeitos no software, que podem causar prejuízos ao negócio do cliente.

6. Estratégia

A estratégia de testes do sistema descreve os objetivos das atividades de teste. Inclui os estágios e os tipos de testes. A estratégia define também a conclusão dos testes e os critérios de sucesso a serem usados.

6.1 Estágios de Teste

Os testes serão distribuídos entre cinco estágios distintos. A seguir citamos esses estágios e descrevemos seus objetivos.

- **Teste de Unidade (TU):** Executado no início da iteração. É aplicado para validar os menores elementos testáveis do software, as classes básicas e os componentes, no modelo de implementação. Verifica se os fluxos de dados e de controle estão funcionando como esperado.

- **Teste de Integração (TI):** È executado para garantir que os componentes do modelo de implementação funcionam apropriadamente quando combinados para executar um *use case*. O objetivo do teste é um pacote ou um conjunto de pacotes do modelo de implementação. O teste de integração revela partes incompletas ou erros na interface entre os pacotes.
  Critérios e testes correspondentes são aplicados a todas as fases as quais são:
  Integridade da interface – interfaces internas e externas são testadas à medida que cada módulo é agregado à estrutura;

Validade Funcional – testes são projetados para descobrir erros funcionais;
Conteúdo Informacional – testes são projetados para descobrir erros associados à estrutura de dados locais ou globais e
Desempenho – testes verificam limites de desempenho estabelecidos durante o projeto.

- **Teste de Sistema (TS):** Denota os aspectos de teste para um subsistema da aplicação. É tradicionalmente feito quando o software está funcionando como um todo. Seu objetivo é a funcionalidade do sistema final. Cuja finalidade principal é exercitar por completo o sistema baseado em computador. É uma série de teste de integração e validação.

- **Teste no Ambiente de Aceitação (TAA):** É normalmente o teste final. O objetivo desse teste é verificar que o software está pronto e pode ser usado pelo usuário final. Ele é realizado pela equipe de desenvolvimento. Deve ser executado num ambiente o mais próximo possível do ambiente de produção. Conhecido como Teste Alfa.

- **Teste de Aceitação (TA):** Teste realizado pelo cliente. Esse teste tem por objetivo a aceitação ou não do sistema por parte do cliente. Depois de realizado com sucesso, o sistema estará pronto para ser implantado no ambiente de produção. Conhecido como o Teste Beta

Mediante as informações acima e considerando a fase em que se encontra o aplicativo e ao curto espaço de tempo, a estratégia de teste a ser adotada será a de Teste de Sistemas.

6.2 Tipos de Teste

- **Teste de Funcionalidade:** Enfatiza a validação das funcionalidades requeridas nos serviços, métodos e *use cases*. Aqui deve ser apresentado a funcionalidade, o desempenho e a confiabilidade e que não apresenta falhas durante o uso normal.

- **Teste de Segurança:** Assegura que os dados e o sistema não serão indevidamente acessados.

- **Teste de Integridade:** Avalia a robustez (resistência à falhas) do sistema, a flexibilidade da linguagem, a sintaxe e a usabilidade do código.

- **Teste de Estresse:** Teste de segurança que avalia como o sistema responde sob condições anormais. Estresse no sistema pode significar excessiva carga de trabalho, insuficiência de memória/serviços ou hardware não disponível. Esse teste é freqüentemente realizado para obter-se um melhor entendimento de como e que áreas do sistema não estão funcionando bem. Com isso planos de contingência e manutenção podem ser orçamentados com antecedência.

- **Teste de Carga:** Avalia a resposta do sistema em condições extremas de carga de informações. Um teste de carga é, na verdade, um teste de performance voltado para a avaliação do sistema em condições extremas ou de limite.

- **Teste de Performance:** Monitora o perfil do tempo, incluindo execução de fluxo, acesso a dados, funcionalidade e chamadas ao sistema para identificar o gargalo e processos ineficazes.

- **Teste de Configuração:** Assegura as funcionalidades pretendidas dos diferentes hardwares e/ou softwares de configuração.

- **Teste de Ciclo de Negócios -** Garante que o sistema funciona apropriadamente durante um ciclo de atividades relativas ao negócio e que ao final desse ciclo todos os resultados esperados foram obtidos.

- **Teste de Interfaces do Usuário -** Assegura que o comportamento, requisitos, projeto gráfico e navegacional definidos para as interfaces sejam atendidos.

- **Teste de Recuperação de Falhas -** Garante que o sistema atende aos requisitos definidos para recuperação de falhas. Normalmente as seguintes situações são abordadas:

    - Falta de energia no cliente;
    - Falta de energia no servidor;
    - Perda da comunicação cliente-servidor;
    - Problemas no sistema operacional;
    - Problemas de endereçamento de memória;
    - Ou qualquer outro erro que faça com que o sistema aborte de forma abrupta sua execução.

  Nestes casos deve ser garantida a consistência das informações e que o sistema retome seu funcionamento normalmente, de preferência retornando ao ponto exato em que estava.

- **Teste de Instalação:** Assegura a instalação, como desejada, nos diferentes *hardwares* e/ou *softwares* de configuração e sob diferentes condições como, por exemplo, espaço em disco insuficiente. *softwares*

- **Teste de Estrutura:** Focaliza a avaliação do design. Normalmente é realizado para aplicações voltadas para a *Web*, garantindo que todos os *links* estejam conectados.

- **Teste de Concorrência:** Valida a capacidade do sistema de lidar com múltiplos atores requisitando recursos (dados, memória) ao mesmo tempo.

- **Teste de Usabilidade:** Focaliza nos fatores humanos, estética, consistência da interface com o usuário, *help on-line* e agentes, manual e material de treinamento. Visa verificar a facilidade que o software possui de ser claramente entendido e facilmente operado pelos usuários. O teste aqui é baseado em *check-list* com vários itens que o software deve atender para que ele possa ser considerado para boa utilização.

6.3 Testes realizados por Estágio

- **Teste de Unidade**
  - Teste de Integridade;
  - Teste de Funcionalidade;
  - Teste de Interface com o Usuário.

- **Teste de Integração**
  - Teste de Integridade;
  - Teste de Funcionalidade;
  - Teste de Interface do Usuário;
  - Teste de Estrutura;
  - Teste de Concorrência;
  - Teste de Usabilidade.

- **Teste de Sistema**
  - Teste de Integridade;
  - Teste de Funcionalidade (realeses);
  - Teste de Ciclo de Negócio;
  - Teste de Interface do Usuário;
  - Teste de Segurança;
  - Teste de Recuperação de Falhas;
  - Teste de Estrutura;
  - Teste de Concorrência;
    - Teste de Sensibilidade;
    - Teste de Desempenho (Estresse);
  - Teste de Usabilidade;
  - Teste de Regressão

- **Teste de Ambiente de Aceitação**
  - Teste de Integridade;
  - Teste de Funcionalidade;
  - Teste de Ciclo de Negócio;
  - Teste de Interface do Usuário;
  - Teste de Performance;
  - Teste de Carga;
  - Teste de Estresse;
  - Teste de Segurança;
  - Teste de Recuperação de Falhas;
  - Teste de Configuração;
  - Teste de Instalação;
  - Teste de Concorrência;
  - Teste de Usabilidade.

- **Teste de Aceitação/homologação**
  - Teste de Integridade;
  - Teste de Funcionalidade;
  - Teste de Ciclo de Negócio;
  - Teste de Interface;
  - Teste de Performance;

- Teste de Carga;
- Teste de Estresse;
- Teste de Segurança;
- Teste de Concorrência;
- Teste de Usabilidade.

Conforme mencionado neste item, o estágio seguido dos seus respectivos testes a ser aplicado neste plano de teste é o Teste de Sistemas. Dentre os testes apresentados serão abordados na Especificação de Casos de Teste de acordo com as necessidades os testes de: Funcionalidade, Interface do Usuário e Segurança.

6.4 Abordagem Utilizada

Realização de Testes – Tarefas

Tarefas	Responsáveis
Planejamento de Teste	Simone M. Cunha
Análise de Teste	Simone M. Cunha
Projeto de Teste	Simone M. Cunha
Implantação de Teste	Simone M. Cunha
Execução de Ensaio	Simone M. Cunha
Avaliação de Teste	Charles Hovadick

Tabela 5 Realização de Testes – Tarefas

Descreve a abordagem de teste utilizada para cada estágio (Ex: manual ou automática, técnicas de teste, etc.) bem como a razão de uso.

Estágio	Técnica/Tipo de Teste	Razão de Uso	Método Utilizado/Técnica	Forma de Comunicação
Teste de Sistemas	Teste de Funcionalidade	Validar se o sistema atende aos requisitos e assegurar que o sistema é confiável.	Particionamento de Equivalência; BVA; CSU; DS; *Realeses*.	Relatório 3 EPCT_Varuna_HD_V.1.0
	Teste de Interface de Usuário	Concentra em ver se a interface de componente se comporta de acordo com sua especificação.	Teste baseado em erro; Teste *Realese*.	Relatório 3 EPCT_Varuna_HD_V.1.0.
	Teste de Segurança	Tem como garantir que	Teste baseado em erro;	Relatório 3 EPCT_Varuna_HD_V.1.0.

		as funções e dados de um sistema sejam acessados apenas por atores autorizados.	Teste *Realese*.	

Tabela 6 Estágios e Técnicas de Teste

As escolhas das técnicas aplicadas vão dependerem do nível o qual o aplicativo se encontra, bem como a apresentação da especificação de requisitos. Assim as técnicas citadas abaixo são as que se aplica ao aplicativo em questão.

Técnicas de Caixa-Preta

Técnicas utilizadas no desenvolvimento de softwares convencionais:

Testes baseados em Grafos:

.Modelagem de fluxo de transação – passos para realizar uma operação, use o DFD.

.Modelagem de estado finito – nós representam as telas abertas em uma operação, as ligações são as transições entre telas. Use o diagrama de estados.

. Modelagem de fluxo de dados – nós são objetos de dados e as ligações são transformações que ocorrem para traduzir um objeto de dados em outro.

Particionamento de Equivalência: avalia uma classe de equivalência para elaborar casos de teste. Conjunto de estados válidos ou inválidos para as condições de entrada que pode ser um valor numérico especifico um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana.

Análise de Valor Limite (BVA): Leva a seleção de casos de testes que trabalham com valores limítrofes (erros que ocorrem nas fronteiras do domínio), seleção de dados nas bordas. O resultado é um teste de limite completo, com maior probabilidade de detecção de erros.

Técnicas utilizadas no desenvolvimento de softwares Orientados a Objetos:

Teste baseado em erro: começa com o modelo de análise e especificação de requisitos. Procura erros nas chamadas de operações ou nas conexões de mensagens. Erros encontrados: Resultado Inesperado uso de operação/mensagem errada, invocação incorreta.
Teste aplicado em atributos e operações (examina o comportamento da operação), focalizando o erro no objeto cliente que chama.
Resultado: pode encontrar um número significativo de erros com esforço relativamente baixo.

Teste de realeses: Para cada teste deve projetar um conjunto de entradas válidas e inválidas e que gere saídas válidas e inválidas. Aqui os cenários mais prováveis devem ser testados primeiro e os cenários incomuns ou excepcionais sejam considerados mais tarde, dedicando as partes mais usadas no sistema. Os casos de uso e diagramas de seqüências podem ser usados durante os testes de integração e de realeses.

6.5 Critérios de Conclusão e Sucesso de Testes

Os testes de unidade são considerados encerrados quando todas as linhas de código forem testadas e nenhum erro for encontrado.

Os demais testes serão considerados concluídos quando:

- 95% dos casos de teste obtiverem sucesso;

N° de Casos de Teste	Casos de Teste com Sucesso	Casos de Teste sem Sucesso	Percentual desejado
13	9	4	12,35

Tabela 7 Critérios de Conclusão

Obs.: Para atingir o critério de aceitação uma taxa de sucesso de pelo menos 95% deve ser atingida antes que o software seja aceito para iniciar o teste de sistema. Então para 10 casos de teste 9,5 devem ser completados.

Durante o teste de sistema, a liberação de novas versões do software deve ser coordenada entre o líder da equipe de desenvolvimento e o analista de teste. A menos que sirvam para corrigir um erro muito grave, novas versões só devem ser liberadas quando objetivos concordados sejam alcançados (por exemplo a próxima versão contém correções para um número 'x' de correções).

7. Registro de Erros Encontrados

Os erros ou problemas encontrados durante os estágios de teste serão formalmente registrados e deverão ser corrigidos imediatamente após serem detectados.

Os *bugs* detectados nos demais estágios deverão ser reportados para o setor **desenvolvimento**.

Erros	Requisitos de Reinicio	Políticas de Teste de Regressão	Critério Conclusão/Suspensão	Critérios de Aprovação/Reprovação
	Descreva todas as políticas relativas à re-execução de testes bem sucedidos quando as configurações do novo teste (corrigir erros ou estender a funcionalidade)	Especificar os critérios (em termos de regressão, as questões em aberto a avaliação dos riscos, e cobertura de código) para a conclusão da execução do teste de	Especificar os critérios (em termos de criticidade dos incidentes de teste) para a suspensão de todos ou em parte da configuração de execução de teste.	Especificar os critérios (em termos de status de objetivos de teste necessários e opcionais) para determinar se uma configuração de teste passou ou falhou.

	transmitidas.	configuração e descrever a conclusão correspondente as técnicas de avaliação.		

Tabela 8 Registro de Erros Encontrados

7.1 Resultado dos Testes

Descrição dos Testes	Implantação do Testware	Relatórios de Teste
Identificar o conjunto de teste de dados (entrada e saída), os procedimentos de teste, teste e o software de suporte personalizado (junto com a documentação do usuário) para ser entregue.		Identificar o log de teste, incidente, status, e relatórios de síntese para ser entregue.

Tabela 9 Resultados de Testes

* Especificar os requisitos de avaliação para cada produto.
Estas observações encontra-se disponíveis no relatório 5 LDT_Varuna_HD_V.1.0.

8. Recursos

Esta seção descreve os recursos humanos, de *hardware* e de *software* necessários para a realização dos testes.

8.1 Recursos Humanos

Nota: as responsabilidades entre o departamento de Garantia de Qualidade de Software (SQA) e o departamento do projeto são as seguintes:
Teste Unitário – responsabilidade da equipe de desenvolvimento;
Teste de sistemas – responsabilidade da garantia de qualidade de software;
Teste de aceitação de usuário – responsabilidade da equipe de representação do usuário;
Teste de conformidade tecnológica – responsabilidade da equipe de instalação e suporte de sistemas.

Teste de Sistemas						
Data	**Quantidade**	**Papel**	**Responsável**	**Status**	**Requisitos**	**Necessidade de Treinamento?**
11/06/2010	1	Arquiteto de Testes	Simone M. Cunha	P	Analista de Teste.	Para esta fase não.
11/06/2010	1	Testador de Sistemas	Simone M. Cunha	P	Analista de Teste	Para esta fase não.

Tabela 10 Recursos Humanos

Status: (P)reenchido.
(E)m (A)Berto.

8.2 Ambiente de Teste (Hardware e Software)

Tipo de Teste	Responsável
Teste de Sistemas	**Simone M. Cunha**

Hardware		
Quantidade	**Tipo**	**Descrição**
1	Servidor *Web*	Perola
1	Servidor de banco de dados	perola

Software	
Tipo	**Descrição**
Browser	IE-Internet Explore.
Ferramenta para acesso ao Banco de Dados	SQL Server 2008
Banco de Dados	SQL Server 2008
Servidor *Web*	
Ferramenta para testes de carga	Não se aplica

Tabela 11 Ambiente de Teste (Hardware e Software)

9. Cronograma

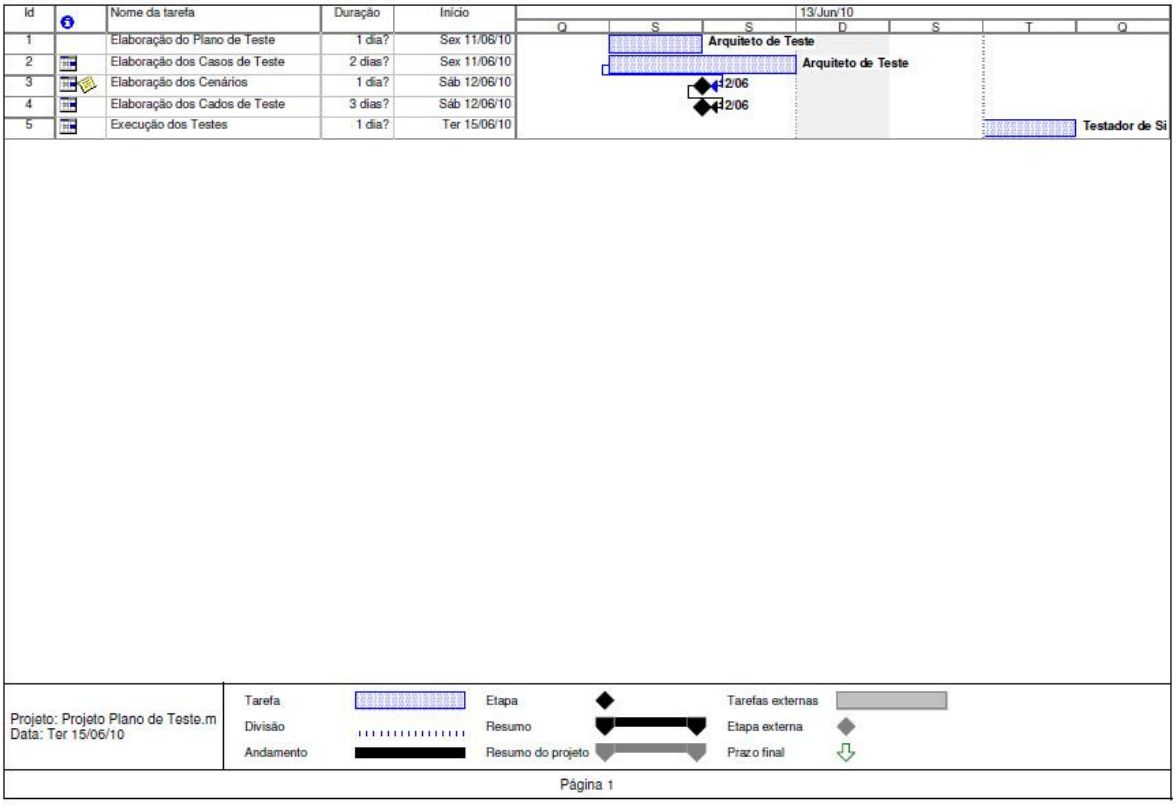


Tabela 12 Cronograma

10. Produtos Entregues

Relação de todos os artefatos que serão gerados pelo processo de testes (Ex: Evidências de teste, casos de teste, relatório dos defeitos encontrados, relatório dos defeitos corrigidos, etc).

1 PTM Plano de Teste Mestre Varuna HD V.1.0;
3 EPCT Especificação e Procedimentos de Caso de Teste Varuna HD V.1.0;
5 LDT Log de Teste Varuna HD V.1.0.

11. Aprovações

Descreve os aprovadores do plano de teste
Aprovamos Plano de Teste Versão revisada 1.1 do projeto **Plano de Teste do sistema HD**.

Simone M. Cunha Analista de Teste	16/06/10	_____
Charles Hovadick Analista Desenvolvedor	16/06/10	_____
Leonardo Carmargos Gerente de Projetos	16/06/10	_____

Varuna Tecnologia Ltda

ESPECIFICAÇÃO E PROCEDIMENTOS DE CASOS DE TESTE

Projeto: Help Desk (HD)

Versão/Revisão 1.0.0.36_01

Responsáveis: Simone Moreira Cunha

Divinópolis – MG

Histórico de Revisão

Data	Versão	Comentário	Autor
11/06/10	1.0	A nomenclatura para o nome desse documento seguirá o seguinte padrão: ordem_nome do artefato_nome do software_sistema_versão do documento.	Simone M. Cunha
17/06/10	1.1	Revisão	Simone M. Cunha

Índice

Índice de Tabelas

1. Introdução

1.1. Propósito

Esse documento contém informações necessárias para verificação dos requisitos do Sistema **Help Desk (HD)**. Casos de testes são utilizados como roteiro para testes de sistema, de aceitação/homologação e integração. A finalidade deste documento é documentar a fase de elaboração dos Casos de Teste (CT) e apresentar o log para cada caso. O log será documentado no relatório: 5 LDT_Varuna_HD_V.1.0.

Teste Caixa-Preta - teste comportamental: categorias de erros a serem detectados:

- ✓ Funções incorretas ou omitidas;
- ✓ Erros de interface;
- ✓ Erros de estrutura de dados ou de acesso a base de dados externa;
- ✓ Erros de comportamento ou desempenho;
- ✓ Erros de iniciação e término.

Este documento atende os padrões da norma IEEE 829 constantes nos seguintes *templates*:

Test Case Specification Template - Especificação de Casos de Teste no item 3;

Test Procedure Specification Template – Especificação de Procedimentos de Teste no item 3;

1.2. Público Alvo

Esse documento destina-se aos envolvidos com a criação, execução e manutenção dos testes.

1.3. Escopo

Neste documento está detalhado o projeto para os casos de teste ao nível de sistema que poderão ser executados de maneira manual ou automática.

1.4. Definições, Acrônimos e Abreviações.

Esta seção descreve definições, acrônimos e abreviações relevantes ao documento.

TC	Test Case (Caso de Teste)
HD	Help Desk
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos

Norma IEEE 829

1.5. Referências

Esta Especificação de Casos de Teste inclui as seguintes referências:

N°	Referência	Versão	Localização	Inclui (Sim/Não)
1	Especificação de Requisitos			Não
2	Sistema de Especificação de Design			Não
3	Detalhe Especificação de Design			Não
4	Guia de Usuário			Não
5	Manuais Operações			Não
6	Guia de Instalação			Não

Para elaboração dos cenários bem como os casos de teste foram utilizados conhecimento da aplicação pelo Arquiteto de Teste.

1.6. Visão geral do documento

- **Na seção** 2 vamos encontrar as regras de atualização dos testes e o mapeamento dos requisitos mostrando quais testes cobrem os requisitos apropriados.
- **Na seção 3** podemos encontrar os casos de teste com suas entradas e procedimentos.

2. Atualização dos testes

Ao atualizar os testes é preciso seguir as seguintes regras de atualização:

- A identificação do caso de teste utiliza a nomenclatura [CTNNN_NI_VV]. Onde NNN é o número do caso de teste, que é único, NI é o nível de um determinado caso de teste. Ao criar novos casos de teste, NNN deve ser igual ao número do maior identificador do documento incrementado de um. VV é a versão do caso de teste. Quando o caso de teste é criado VV é 00, a medida que o caso de teste é modificado ele assume os valores 01, 02, e assim por diante;
- A seção **3** deve estar sempre consistente. Todo caso de teste criado ou atualizado deve estar na coluna direita da Tabela 1 e mapeado com algum requisito da coluna esquerda. Quando o caso de teste for removido ele deve ser retirado da Tabela 1. Mesmo se um requisito não contiver um caso de teste associado o mesmo deve permanecer na próxima seção;

A tabela abaixo contém os testes mapeados para os requisitos encontrados.

Identificação do Requisito	Identificação do Caso de Teste
Login	CT_001_00_00
	CT_001_01_00
Cadastro de Relatório de Visita Técnica	CT_002_00_00
Dados do RVT	
Cadastro de Participante (s)	CT_003_00_00
Consulta de Relatório de Visita Técnica	CT_004_00_00
Cadastro de Chamados	CT_005_00_00
Consulta de Chamados	CT_006_00_00
Gerenciamento de Tarefas do Colaborados:	CT_007_00_00
	CT_007_01_00
Tarefas Pendentes;	CT_007_02_00
Tarefas Iniciadas;	
Tarefas Paralisadas.	
Cadastro de Cliente	CT_008_00_00
Validação CPF/CNPJ	CT_008_01_00
Consulta de Cliente	CT_009_00_00
Cadastro de Relatório On Line	CT_010_00_00
Cadastro Módulo	CT_010_01_00
Consulta de Relatório On Line	CT_011_00_00
Cadastro de Atualização Remota de Banco de Dados	CT_012_00_00

Consulta de Atualização Remota de Banco de Dados	CT_013_00_00

Tabela 1. Rastreamento dos Requisitos

Nome do Cenário de Teste:	**Login Inválido**
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Acessar o endereço: http://varuna.gutix.com.br:8085/varuna/VarunaAtendimento/forms/login.aspx. 2) Informar o Usuário Inválido. 3) Informar uma Senha Válida. 4) Clicar em 'Ok'.	CT_001_00_00 Usuário Inválido e Senha válida Usuário válido e Senha inválida CT_001_01_00

Tabela 2 Elaboração do Cenário de Teste Login

Nome do Cenário de Teste:	**Cadastro de Relatório de Visita Técnica**
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Relatório de Visita Técnica'. 2) Clicar no botão 'inserir'. 3) Informar os demais campos obrigatórios. 4) Clicar em 'Salvar'.	CT_002_00_00 Cadastro de RVT

Tabela 3 Elaboração do Cenário de Teste Cadastro de Relatório de Visita Técnica

Nome do Cenário de Teste:	Cadastro de Participantes
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Participantes' 2) Clicar no botão 'inserir'. 3) Informar os demais campos obrigatórios. 4) Clicar em 'Ok'.	CT_003_00_00 Cadastro de Participantes

Tabela 4 Elaboração do Cenário de Teste Cadastro de Participantes

Nome do Cenário de Teste:	Consulta de Relatório de Visita Técnica
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Consulta'. 2) Passar um dos parâmetros. 3) Clicar em 'Pesquisar'.	CT_004_00_00 Consulta de RVT

Tabela 5 Elaboração do Cenário de Teste Consulta de Relatório de Visita Técnica

Nome do Cenário de Teste:	Cadastro de Chamado
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Chamado'. 2) Clicar em 'Inserir'. 3) Informar os campos obrigatórios. 4) Clicar em 'Salvar'.	CT_005_00_00 Cadastro de Chamado

Tabela 6 Elaboração do Cenário de Teste Cadastro de Chamado

Nome do Cenário de Teste:	Consulta de Chamado
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Consulta'. 2) Passar os parâmetros de pesquisa. 3) Clicar em 'Pesquisar'.	CT_006_00_00 Consulta de Chamado

Tabela 7 Elaboração do Cenário de Teste Consulta de Chamado

Nome do Cenário de Teste:	Gerenciamento Tarefas do Colaborador
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Tarefas – Gerenciamento'. 2) Em 'Tarefas Pendentes' clicar em 'Iniciar Execução da Tarefa'. 3) Em 'Tarefas Iniciadas' clicar em 'Comentar Tarefa'. Clicar em 'Finalizar/Parar Tarefa'. 4) Em 'Tarefas Paralisadas' clicar em 'Tarefas Pausadas'. Clicar em 'Reiniciar Execução da tarefa', clique em 'Ok'.	CT_007_00_00 Iniciar Execução de tarefas pendentes CT_007_01_00 Gerenciar tarefas iniciadas CT_007_02_00 Gerenciar tarefas paralizadas

Tabela 8 Elaboração do Cenário de Teste Gerenciamento Tarefas do Colaborador

Nome do Cenário de Teste:	Cadastro de Cliente
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Cadastro de Cliente'. 2) Clicar no botão 'Inserir'. 3) Informar o CPF/CNPJ. 4) Informar os demais campos obrigatórios. 5) Clicar em 'Salvar'.	CT_008_00_00 Preenchimento dos campos obrigatórios na tela CT_008_01_00 Validação de CPF/CNPJ

Tabela 9 Elaboração do Cenário de Teste Cadastro de Cliente

Nome do Cenário de Teste:	Consulta de Cliente
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Consulta'. 2) Informar os parâmetros para consulta. 3) Clicar em 'Pesquisar'.	CT_009_00_00 Consulta Cliente

Tabela 10 Elaboração do Cenário de Teste Consulta de Cliente

Nome do Cenário de Teste:	Cadastro de Relatório On Line
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Publicação On Line de Relatório'. 2) Clicar no botão 'Inserir'. 3) Informar os campos obrigatórios. 4) Anexar o arquivo correspondente ao relatório. 5) Clicar em Salvar.	CT_10_00_00 Cadastro de Relatório On Line
6) Abrir a tela 'Módulo'. 7) Clicar em 'Inserir'. 8) Infomar o módulo o qual quer adicionar.	CT_10_01_00 Cadastro de Módulo

Tabela 11 Elaboração do Cenário de Teste Cadastro de Relatório On Line

Nome do Cenário de Teste:	Consulta de Relatório
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Consulta'. 2) Informar os parâmetros de consulta. 3) Clicar em 'Pesquisar'.	CT_011_00_00 Consulta de Relatório

Tabela 12 Elaboração do Cenário de Teste Consulta de Relatório

Nome do Cenário de Teste:	Atualização Remota de Banco de Dados
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Cadastro de Atualização Remota de Banco de Dados'. 2) Clicar em 'Inserir'. 3) Informar os campos obrigatórios. 4) Clicar em 'Salvar'.	CT_012_00_00 Cadastro de Atualização Remota de Banco de Dados

Tabela 13 Elaboração do Cenário de Teste Atualização Remota de Banco de Dados

Nome do Cenário de Teste:	Consulta de Atualização Remota de Banco de Dados
Procedimentos necessários para execução do cenário de testes.	**Identificação do Caso de Teste**
1) Abrir a tela 'Consulta'. 2) Informar os parâmetros de consulta. 3) Clicar em 'Pesquisar'.	CT_013_00_00 Consulta de Relatório

Tabela 14 Elaboração do Cenário de Teste Consulta de Atualização Remota de Banco de Dados

3. Casos de Teste

3.1. [CT001_00_00] – Login Usuário Inválido e Senha válida

Caso de Teste:	CT_001_00_00 Login Usuário e Senha Inválidos	
Descrição	Objetivo inserir um usuário inválido e senha válida com intuito de realizar o Teste de Segurança. Condições especiais: não se aplica.	
Pré-Condições	O Tipo de Autenticação deve está selecionada, se Windows ou SQLServer.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Acessar o endereço: http://varuna.gutix.com.br:8085/varuna/VarunaAtendimento/forms/login.aspx; Inserir um usuário inválido com uma senha válida; Clicar em Ok.	Usuário sem acesso ao sistema.
Dados de Entrada	Tipo de Autenticação: SqlServer (por padrão este item vêem selecionado) Usuário não cadastrado neste domínio	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.2. [CT001_01_00] – Login Usuário Válido e Senha inválida

Caso de Teste:	CT_001_01_00 Login Usuário e Senha válidos	
Descrição	Objetivo inserir um usuário válido e senha inválida com intuito de realizar o Teste de Segurança. Condições especiais: não se aplica.	
Pré-Condições	O Tipo de Autenticação deve está selecionada, se Windows ou SQLServer.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Acessar o endereço: http://varuna.gutix.com.br:8085/varuna/VarunaAtendimento/forms/login.aspx; Inserir um usuário válido com uma senha inválida; Clicar em Ok.	Usuário sem acesso ao sistema.
Dados de Entrada	Tipo de Autenticação: SqlServer (por padrão este item vêem selecionado) Usuário não cadastrado neste domínio	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.3. [CT002_00_00] – Cadastro de Relatório de Visita Técnica

Caso de Teste:	CT_002_00_00 Cadastro de Relatório de Visita Técnica	
Descrição	Objetivo inserir as Visitas Técnicas feitas nos clientes. Condições especiais: não se aplica.	
Pré-Condições	Os campos: Unidade de Negócio, Tipo de Visita, Colaborador e Cliente já devem está pré-cadastrados.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Abrir a tela 'Cadastro'; Clicar no botão 'Inserir'; Informar os campos obrigatórios; Clicar em 'Salvar'	Inserção de RVT's.
Dados de Entrada	Campos Obrigatórios	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.4. [CT003_00_00] – Cadastro de Participantes

Caso de Teste:	CT_003_00_00 Cadastro de Participantes	
Descrição	Objetivo inserir os participantes para serem associados aos Relatórios de Visitas Técnicas feitas nos clientes. Condições especiais: não se aplica.	
Pré-Condições	O RVT já deve está pré-cadastrado.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Abrir a tela 'Participante'; Clicar no botão 'Inserir'; Informar os campos obrigatórios; Clicar em 'Ok'	Inserção de participantes.
Dados de Entrada	Campos Obrigatórios	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.5. [CT004_00_00] – Consulta RVT

Caso de Teste:	CT_004_00_00 Consulta RVT	
Descrição	Objetivo consultar os RVT's já cadastrados. Condições especiais: não se aplica.	
Pré-Condições	Antes de clicar no botão 'Pesquisar' parâmetros devem ser informados.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Abrir a tela 'Consulta'; Informar os parâmetros de pesquisa desejados; Clicar em 'Pesquisar'	Dados de acordo com os parâmetros de pesquisa informados.
Dados de Entrada	Parâmetros de Pesquisa	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.6. [CT005_00_00] – Cadastro de Chamado

Caso de Teste:	CT_005_00_00 Cadastro de Chamado	
Descrição	Objetivo cadastrar chamados feitos pelos clientes para controle do atendimento. Condições especiais: não se aplica.	
Pré-Condições	Os campos: 'Tipo de Chamado', 'Cliente', 'Módulo' e 'Setor Responsável' devem estar pré-cadastrados para serem informados no momento do cadastro do Chamado.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Abrir a tela 'Chamado'; Clicar no botão 'Inserir'; Informar os campos obrigatórios; Clicar em 'Salvar'	Inserção do Chamado de acordo com os dados informados.
Dados de Entrada	Todos os campos obrigatórios.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.7. [CT006_00_00] – Consulta de Chamado

Caso de Teste:	CT_006_00_00 Cadastro de Chamado	
Descrição	Objetivo consultar os chamados cadastrados. Condições especiais: não se aplica.	
Pré-Condições	Antes de clicar no botão 'Pesquisar' parâmetros devem ser informados.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Abrir a tela 'Consulta'; Informar os parâmetros de pesquisa desejados; Clicar em 'Pesquisar'	Dados de acordo com os parâmetros de pesquisa informados.
Dados de Entrada	Parâmetros de Pesquisa.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.8. [CT007_00_00] – Gerenciamento de Tarefas Pendentes

Caso de Teste:	CT_007_00_00 Gerenciamento de Tarefas	
Descrição	Objetivo gerenciar as tarefas pendentes originadas pelos chamados cadastrados vinculados a um determinado colaborador. Condições especiais: não se aplica.	
Pré-Condições	É necessário que um chamado esteja liberado para atendimento.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Na tela 'Consulta de Chamado', pesquisar o respectivo chamado a ser liberado. Clique em 'Liberação de Chamado'; Escolha a prioridade, associe a um colaborador, marque a opção 'Liberar este chamado' = 'Sim', clique em 'Confirmar'. Abrir a tela 'Gerenciamento de Tarefas do Colaborador'; Clicar em 'Iniciar Execução da Tarefa'; Clicar em 'Ok'.	A respectiva tarefa é redirecionada para a guia de tarefas iniciadas.
Dados de Entrada	Informações necessárias para gerenciamento da chamada.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.9. [CT007_01_00] – Gerenciamento de Tarefas Iniciadas

Caso de Teste:	CT_007_01_00 Gerenciamento de Tarefas	
Descrição	Objetivo gerenciar as tarefas iniciadas originadas das tarefas pendentes. Condições especiais: não se aplica.	
Pré-Condições	.Ter Iniciado a Execução da Tarefa.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Abrir a tela 'Gerenciamento de Tarefas do Colaborador'; Na guia especifica clicar em 'Comentar Tarefa' caso queira colocar alguma observação com relação à tarefa; Clicar em 'Finalizar/Paralisar a tarefa'.	Documentar a respectiva tarefa caso seja necessário. Finalizar ou paralisar a respectiva tarefa.
Dados de Entrada	Informações necessárias para gerenciamento da chamada.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.10. [CT007_02_00] – Gerenciamento de Tarefas Paralizadas

Caso de Teste:	CT_007_02_00 Gerenciamento de Tarefas	
Descrição	Objetivo gerenciar as tarefas paralisadas originadas das tarefas Iniciadas. Condições especiais: não se aplica.	
Pré-Condições	Clicar em 'Finalizar/Paralisar a tarefa'.	
Procedimento/Resultado Esperado	**Procedimento**	**Resultado Esperado**
	Abrir a tela 'Gerenciamento de Tarefas do Colaborador'; Na guia especifica clicar em 'Finalizar/Paralisar a tarefa'. Escolher a opção: 'Paralisa';	A tarefa deverá ser redirecionada para a guia 'Tarefas Paralisadas'.
Dados de Entrada	Informações necessárias para gerenciamento da chamada.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.11. [CT008_00_00] – Cadastro de Clientes

Caso de Teste:	CT_008_00_00 Cadastro de Clientes	
Descrição	Objetivo inserir um cliente validando os campos obrigatórios. Condições especiais: não se aplica.	
Pré-Condições	Preenchimento dos campos com asterisco vermelho é obrigatório.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Cadastro de Cliente'; Clicar no Botão 'Inserir'; Inserir os campos obrigatórios indicados por um asterisco vermelho; Clicar no botão 'Salvar'.	Inserção do cadastro do cliente.
Dados de Entrada	Todos os campos obrigatórios.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.12. [CT008_01_00] – Validação do campo 'CPF/CNPJ'

Caso de Teste:	CT_008_01_00 Validação do campo 'CPF/CNPJ'	
Descrição	Objetivo validar o campo 'CPF/CNPJ. Condições especiais: não se aplica.	
Pré-Condições	Informar um CPF ou CNPJ inválido.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Cadastro de Cliente'; Clicar no Botão 'Inserir'; Inserir os campos obrigatórios indicados por um asterisco vermelho, inclusive o campo 'CPF/CNPJ'; Clicar no botão 'Salvar'.	Retornar uma mensagem: 'CPF/CNPJ' Inválido.
Dados de Entrada	'CPF/CNPJ' inválido.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.13. [CT009_00_00] – Consulta Cliente

Caso de Teste:	CT_009_00_00 Consulta Cliente	
Descrição	Objetivo consultar os dados de um cliente cadastrado. Condições especiais: não se aplica.	
Pré-Condições	Parâmetros de pesquisa devem ser informados antes de clicar em 'Pesquisa'.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Consulta de Cliente'; Inserir os parâmetros necessários para a pesquisa; Clicar 'Pesquisa'.	Retornar dados de acordo com os parâmetros passados.
Dados de Entrada	Parâmetros necessários para pesquisa.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.14. [CT010_00_00] – Cadastro de Relatório

Caso de Teste:	CT_010_00_00 Cadastro de Relatório	
Descrição	Objetivo cadastrar relatórios para serem publicados de forma on line no cliente. Condições especiais: não se aplica.	
Pré-Condições	'Tipo de Relatório' deve está pré-cadastrado.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Cadastro'; Inserir os campos obrigatórios; Clicar 'Salvar'.	Cadastro de relatórios.
Dados de Entrada	Campos obrigatórios.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.15. [CT010_01_00] – Cadastro de Módulo

Caso de Teste:	CT_010_01_00 Cadastro de Módulo	
Descrição	Objetivo cadastrar os módulos que serão associados aos relatórios cadastrados. Condições especiais: não se aplica.	
Pré-Condições	O relatório já deve está cadastrado.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Cadastro'; Inserir os campos obrigatórios; Clicar 'Salvar'.	Cadastro de módulos.
Dados de Entrada	Campos obrigatórios.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.16. [CT011_00_00] – Consulta Relatório

Caso de Teste:	CT_011_00_00 Consulta de Relatório	
Descrição	Objetivo visualizar os relatórios cadastrados. Condições especiais: não se aplica.	
Pré-Condições	Informar os parâmetros necessários a consulta antes de clicar em 'Pesquisar'.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Consulta'; Inserir os parâmetros necessários; Clicar 'Pesquisar'.	Visualização de relatórios cadastrados.
Dados de Entrada	Parâmetros necessários para consulta.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.17. [CT012_00_00] – Cadastro de atualização remota de Banco de Dados

Caso de Teste:	CT_012_00_00 Cadastro de atualização remota de Banco de Dados.	
Descrição	Objetivo cadastrar os scripts para atualização do banco de dados respectivo a aplicação. Condições especiais: não se aplica.	
Pré-Condições	Inserir os campos obrigatórios.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Cadastro'; Clicar em 'Inserir'; Inserir os campos obrigatórios; Clicar em 'Salvar'.	Inserção dos scripts para atualização do banco de dados.
Dados de Entrada	Campos obrigatórios.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

3.18. [CT013_00_00] – Consulta de atualização remota de Banco de Dados

Caso de Teste:	CT_013_00_00 Consulta de atualização remota de Banco de Dados.	
Descrição	Objetivo consultar os scripts para atualização do banco de dados respectivo a aplicação já cadastrados. Condições especiais: não se aplica.	
Pré-Condições	Informar os parâmetros de pesquisa antes de clicar.	
Procedimento/Resultado Esperado	Procedimento	Resultado Esperado
	Abrir a tela de 'Consulta'; Inserir os parâmetros de pesquisa; Clicar em 'Pesquisar'.	Visualização dos scripts cadastrados.
Dados de Entrada	Parâmetros de pesquisa.	
Critérios Especiais	Não se aplica	
Ambiente	Item informado no Plano de Teste (1 PTM).	
Implementação	Manual	
Iteração	1.0.0.36	

Varuna Tecnologia Ltda

LOG DE TESTE
Projeto: Help Desk (HD)

Versão/Revisão 1.0.0.36_01

Responsáveis: Simone Moreira Cunha

Divinópolis – MG

Histórico de Revisão

Data	Versão	Comentário	Autor
15/06/10	1.0	A nomenclatura para o nome desse documento seguirá o seguinte padrão: ordem_nome do artefato_nome do software_sistema_versão do documento.	Simone M. Cunha
18/06/10	1.1	Revisão	Simone M. Cunha

Índice

Índice de Tabelas

Introdução

Propósito

- ✓ Esse documento contém informações necessárias sobre os resultados da execução dos casos de teste.

Atende os padrões da norma IEEE 829.

Público Alvo

Esse documento destina-se aos envolvidos com a execução e manutenção dos testes.

Escopo

Neste documento está detalhado o resultado da aplicação dos casos de teste ao nível de sistema que poderão ser executados de maneira manual ou automática.

Definições, Acrônimos e Abreviações.

Esta seção descreve definições, acrônimos e abreviações relevantes ao documento.

HD	Help Desk
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
Msg	Mensagem
TC	Test Case (Caso de Teste)

3.2 Log de Teste

3.2.1 Referências

Este Log de Teste inclui as seguintes referências:

N°	Referência	Versão	Localização	Inclui (Sim/Não)
1	Especificação de Caso de Uso			Não
2	Especificação de Processo			Não
3	Relatório de Transmissão			Não

Tabela 1 Referências

3.2.2 Entrada de Atividade e Evento

Nome do Caso de Teste	CT_001_00_00 Login usuário inválido e senha válidos

Data	**Hora**		**Responsável**	
	Início	**Fim**		
12/06/2010	21:20	21:35	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Acessar o link abaixo; Informar um usuário inválido; Informar uma senha válida; Clicar no botão 'Ok'.	Vide anexo	S	Não se aplica.	Não se aplica

http://varuna.gutix.com.br:8085/varuna/VarunaAtendimento/forms/login.aspx

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Anexo de Resultado


HelpDesk (HD)

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_001_01_00 Login usuário válida e senha inválida

Data	Hora		Responsável	
	Início	Fim		
12/06/2010	21:40	21:51	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Acessar o link abaixo; Informar um usuário válido; Informar uma senha inválida; Clicar no botão 'Ok'.	Vide anexo	S	Não se aplica.	Não se aplica

http://varuna.gutix.com.br:8085/varuna/VarunaAtendimento/forms/login.aspx

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Anexo de Resultado


Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_002_00_00 Cadastro de RVT's

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	21:53	21:56	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Cadastro de RVT'; Clicar no Botão 'Inserir'; Inserir os campos obrigatórios indicados por um asterisco vermelho; Clicar no botão 'Salvar'.	Dados cadastrados com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_003_00_00 Cadastro de Participantes

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	21:57	21:59	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Participantes'; Clicar em 'Inserir'; Informar os		S	Não se aplica.	Não se aplica

campos obrigatórios; Clicar 'Ok'.				

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_004_00_00 Consulta de RVT

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	21:59	21:59	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Consulta'; Informar os parâmetros para pesquisa; Clicar no botão 'Pesquisar'.	Consulta realizada com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_005_00_00 Cadastro de Chamados

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:03	22:06	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Chamado'; Clicar em 'Inserir'; Informar os campos obrigatórios; Clicar 'Salvar'.	Cadastro realizado com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

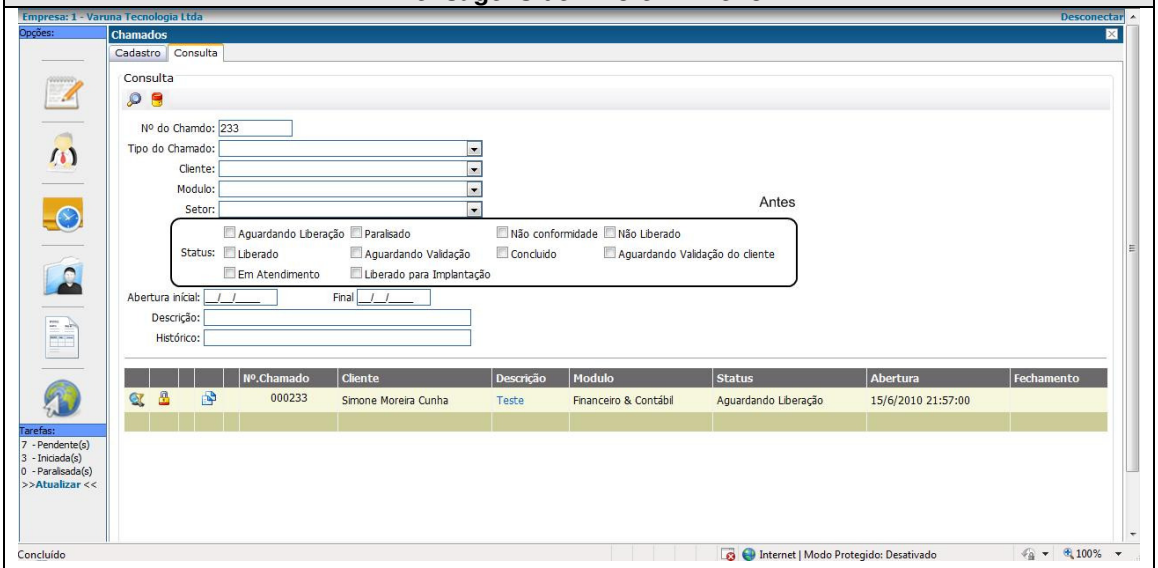
Nome do Caso de Teste	CT_006_00_00 Consulta de Chamado

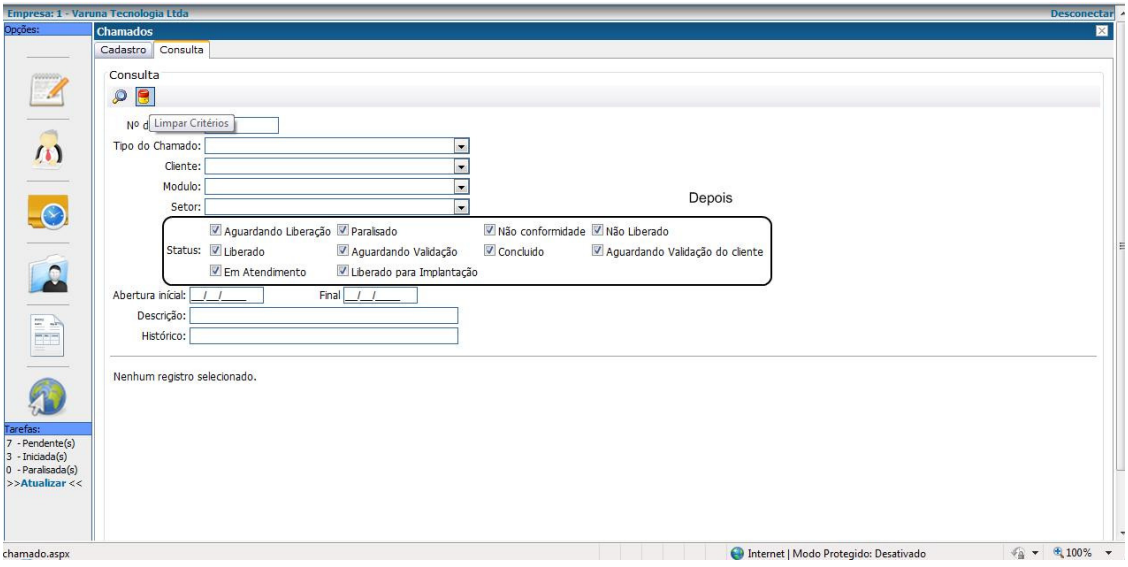
Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:06	22:17	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Consulta'; Informar os parâmetros para pesquisa; Clicar no botão 'Pesquisar'.		S	Não se aplica.	Não se aplica
Teste do botão 'Limpar Critérios' da tela 'Consulta Chamados'.	Vide msg em anexo.	D	Não se aplica.	Não se aplica
Todas as opções de	Dados coerentes ao	S		

status foram testadas.	item selecionado para consulta.			

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo





Categoria do Erro	A	B	C
			X

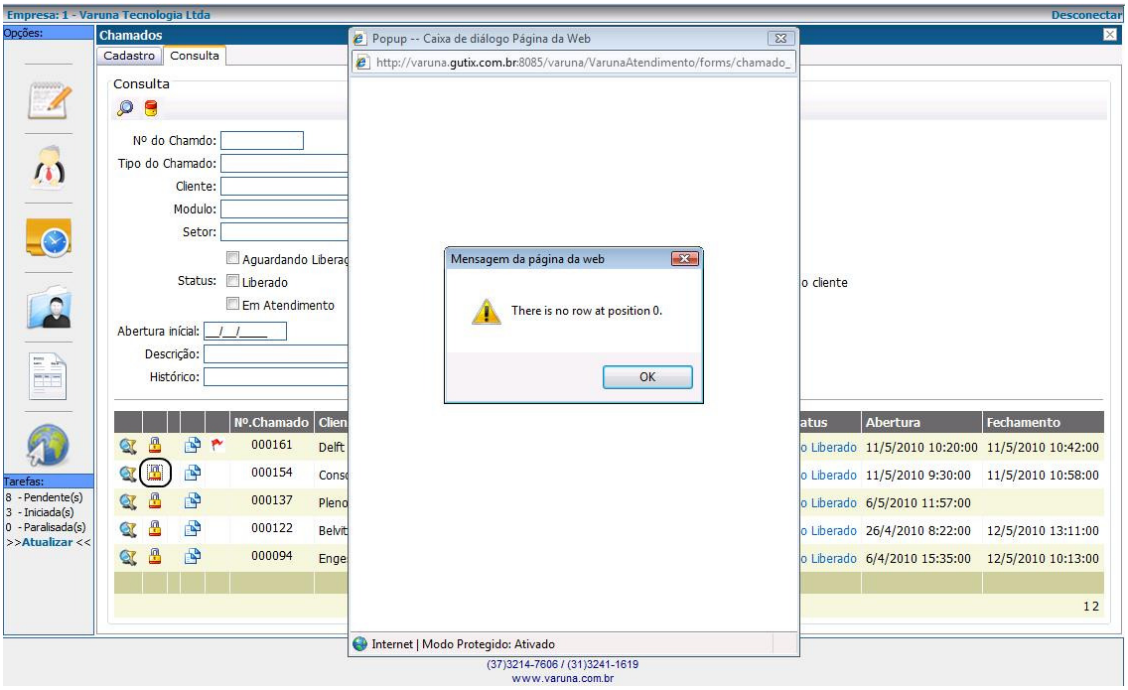
Nome do Caso de Teste	CT_007_00_00 Gerenciamento de tarefas do colaborador/Tarefas Pendentes

Data	**Hora**		**Responsável**	
	Início	**Fim**		
15/06/2010	22:18	22:27	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Na tela 'Consulta de Chamado', pesquisar o respectivo chamado a ser liberado. Clique em 'Liberação de Chamado'; Escolha a prioridade, associe a um colaborador, marque a opção 'Liberar este chamado' = 'Sim', clique em 'Confirmar'.	F1 - Vide msg de erro em anexo ao clicar em 'Liberação de Chamado'. F2 – Veja que está faltando a opção de confirmação na tela	D		
Abrir a tela Gerenciamento de tarefas do colaborador; Na guia 'Tarefas Pendentes', visualize a tarefa a ser executada, clique em 'Iniciar Execução da Tarefa', em seguida clique em 'Ok'.	Tarefa foi enviada para a aba 'Tarefas Iniciadas.	S	Não se aplica.	Não se aplica

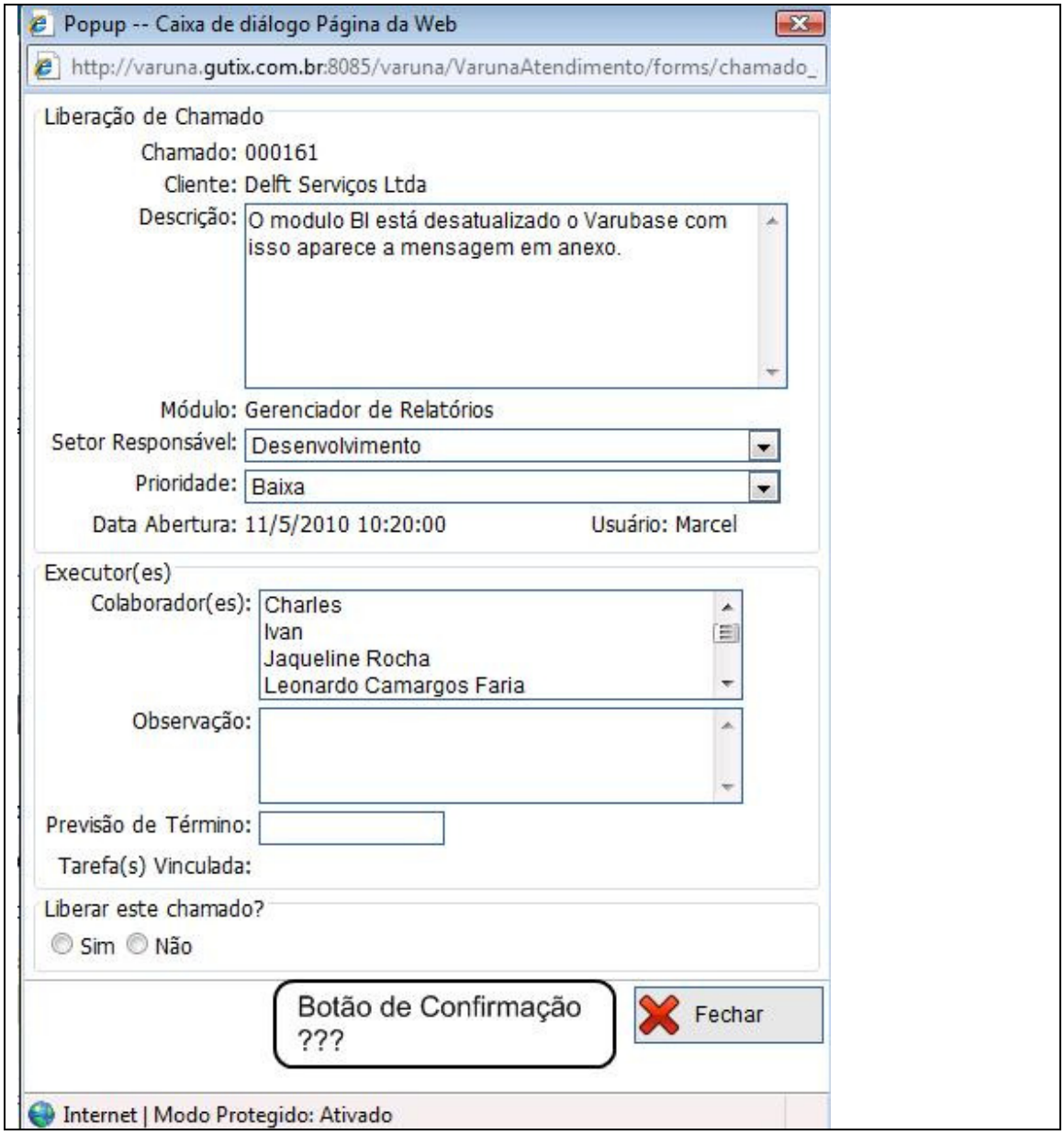
Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

F1



F2



Categoria do Erro	A	B	C
	x		

Nome do Caso de Teste	CT_007_01_00 Gerenciamento de tarefas do colaborador/Tarefas Iniciadas

Data	**Hora**		**Responsável**	
	Início	**Fim**		
15/06/2010	22:28	22:39	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
- Abrir a tela Gerenciamento de tarefas do colaborador; Na guia 'Tarefas Iniciadas', visualize a tarefa a ser comentada e clique em 'Comentar Tarefa', clique em 'Inserir', digite o comentário, clique em 'Confirmar'. - Clique em 'Finalizar/Paralisar Tarefa', informe o status 'Pausadas', digite uma obs, clique em 'Confirmar'. Em 'Tarefas Iniciadas', clique em 'Finalizar/Paralisar', informe o status 'Finalizar' e clique em 'Confirmar'.	Tarefa comentada com sucesso. Tarefa enviada para a guia 'Tarefas Paralisadas'. Tarefa concluída retornada a tela de 'Chamados' com status 'Aguardando Validação'.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	**A**	**B**	**C**

Nome do Caso de Teste	CT_007_02_00 Gerenciamento de tarefas do colaborador/Tarefas Paralisadas

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:39	22:44	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela Gerenciamento de tarefas do colaborador; Em 'Tarefas Pausadas', clique em 'Reiniciar Execução da tarefa', clique em 'Ok.	Tarefa enviada para a guia 'Tarefas Iniciadas.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_008_00_00 Cadastro de Clientes Preenchimento dos campos obrigatórios na tela

Data	Hora		Responsável	
	Início	Fim		
12/06/2010	20:19	20:28	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Cadastro de Cliente'; Clicar no Botão 'Inserir'; Inserir os campos	F1 - Vide msg de erro em anexo. O campo 'Número' não está marcado com asterisco	F	Não se aplica.	Não se aplica

obrigatórios indicados por um asterisco vermelho; Clicar no botão 'Salvar'.	para informar que é um campo obrigatório. De certa forma este campo não deve ser, uma vez que existe endereços que não possui número, usando a nomenclatura s/n.			

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo	
	

Categoria do Erro	**A**	**B**	**C**
	X		

Nome do Caso de Teste	CT_008_01_00 Cadastro de Clientes Validação CPF/CNPJ

Data	**Hora**		**Responsável**	
	Início	**Fim**		
15/06/2010	22:44	22:46	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Cadastro de Cliente'; Clicar no Botão 'Inserir'; Inserir os campos	F1 – Vide msg em anexo. Sistema permitiu inserção.	F	Não se aplica.	Não se aplica

obrigatórios indicados por um asterisco vermelho; Informar o CPF/CNPJ inválido; 1º Teste: CPF com 9 dígitos. Clicar no botão 'Salvar'.				
Abrir a tela de 'Cadastro de Cliente'; Clicar no Botão 'Inserir'; Inserir os campos obrigatórios indicados por um asterisco vermelho; Informar o CPF/CNPJ inválido; 2º Teste: CPF com 12 dígitos. Clicar no botão 'Salvar'.	F1 – Vide msg em anexo. Sistema permitiu inserção.	F		

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

F1

Categoria do Erro	A	B	C
	x		

Nome do Caso de Teste	CT_009_00_00 Consulta de Clientes

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:47	22:50	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Consulta de Cliente'; Informar os parâmetros de pesquisa; Clicar em 'Pesquisar'.	Pesquisa retornada com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_010_00_00 Cadastro de Relatório

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:49	22:50	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Cadastro de Relatório'; Clicar no Botão 'Inserir'; Inserir os campos obrigatórios; Clicar no botão 'Salvar'.	Relatório cadastrado com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;

(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_010_01_00 Cadastro de Módulo

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:51	22:53	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de Cadastro de Relatório/Módulo; Clique em 'Adicionar'; Informe o módulo a ser adicionado; Clique em 'Confirmar'.	Módulo cadastrado/associado com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_011_00_00 Consulta de Relatório

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:54	22:58	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de 'Consulta de Relatório';	Pesquisa retornada com sucesso.	S	Não se aplica.	Não se aplica

Informar os parâmetros de pesquisa; Clicar em 'Pesquisar'.				

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_012_00_00 Cadastro de Atualização Remota de Banco de Dados

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:57	22:59	Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de Cadastro de 'Atualização Remota de Banco de Dados'; Clique em 'Inserir'; Informe os campos obrigatórios; Clique em 'Salvar'.	Script cadastrado com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

Nome do Caso de Teste	CT_013_00_00 Consulta de atualização remota

Data	Hora		Responsável	
	Início	Fim		
15/06/2010	22:56		Simone M. Cunha	
Procedimentos	**Resultados**	**Status**	**Informações do ambiente**	**Eventos/anomalias**
Abrir a tela de Consulta de 'Atualização Remota'; Informar os parâmetros de pesquisa; Clicar em 'Pesquisar'.	Pesquisa retornada com sucesso.	S	Não se aplica.	Não se aplica

Status: (S)ucesso;
(F)racasso;
(D)esconhecido.

Mensagens de Erro em Anexo

Categoria do Erro	A	B	C

3.3 Incidente de Teste

Gerenciamento de Erro e de Configuração

Destinado à: Equipe de revisão de erros

Erros concordados como válidos serão categorizados pela equipe de revisão de erros da seguinte forma:

Categoria	**Descrição**	**Prioridade** (Devem ser eliminados em:)
A	Erros sérios que impeçam a continuidade do teste de sistema de uma função em particular, ou graves erros de dados.	48 horas (a contar da hora em que o erro foi levantado até a correção a ser liberada no ambiente de teste) pela equipe de correção de bugs. Caso o erro impeça o teste de sistema de continuar, a eliminação deve ser dentro de 4 horas.

B	Erros relativos a dados sérios ou faltantes que não impeçam a implantação.	1 dia.
C	Pequenos erros que não impeçam e nem afetam a funcionalidade.	3 dias.

Liberação de novas versões

A liberação de novas versões de software deve ser coordenada com o analista de teste – novas versões devem ser somente liberadas quando concordadas, e onde haja um benefício definitivo (e.g que contenha correções para um número X de bugs).