CÉSAR AUGUSTO SILVESTRINI ISHIDA

APLICAÇÕES DE TÉCNICAS DE TESTE DE SOFTWARE EM DIFERENTES ARQUITETURAS DE SISTEMAS

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para conclusão do Curso de MBA em Engenharia de Software.

São Paulo
2003

ESCOLA POLITECNICA DA USP

CÉSAR AUGUSTO SILVESTRINI ISHIDA

APLICAÇÕES DE TÉCNICAS DE TESTE DE SOFTWARE EM DIFERENTES ARQUITETURAS DE SISTEMAS

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para conclusão do Curso de MBA em Engenharia de Software.

Área de Concentração:
Engenharia de Software

Orientação:
Prof. Dr. Kechi Hirama

São Paulo
2003

ESP/ES
2003
I3a

FICHA CATALOGRÁFICA

Ishida, César Augusto Silvestrini
Aplicações de Técnicas de Teste de Software em Diferentes Arquiteturas de Sistemas.
São Paulo, 2002. 70 p.
Dissertação (MBA) - Escola Politécnica da Universidade de São Paulo.
1. Teste de Software.

M2003K SYSNO: 1418976

RESUMO

O presente trabalho tem o propósito de descrever as mais utilizadas técnicas de teste de software às principais arquiteturas de sistemas utilizadas no mercado, propondo qual técnica seria melhor aplicada a determinado tipo de sistema. Este estudo foi realizado a partir de pesquisas das técnicas de software mais conhecidas e também de pesquisas das arquiteturas de sistemas. Para relação entre técnica de teste de software e os sistemas foi realizado um estudo com base em opiniões de diferentes autores e artigos. A solução proposta não é única e definitiva, mas sim, um guia para testar diferentes arquiteturas de software.

ABSTRACT

The present work has the intention to describe the most used techniques of software testing to the main system architectures used in the market, considering which technique better would be applied the determined type of system. This study it was carried through from research of the known techniques of software and, also, a research of the system architectures. For relation between technique of software testing and the systems a study of opinions of different authors and articles were carried through. The present solution is not definitive but, It is a guide to test different software architectures.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

Neste primeiro capítulo do trabalho será descrito qual é o objetivo do estudo realizado, quais foram as motivações para a escolha e o posterior aprofundamento do tema e por fim, como será a estrutura de capítulos do trabalho.

1.1. Objetivo

O objetivo deste trabalho é apresentar como os principais tipos de método de testes existentes se aplicam nas diferentes formas de arquitetura de sistemas utilizadas no desenvolvimento de sistemas de software.

Para isso, primeiramente, são descritas as diferentes técnicas de teste de Software focando em seus conceitos, processos de desenvolvimento e aplicações.

Logo após, segue a exposição das diferentes arquiteturas de sistemas onde, novamente, para cada uma, serão focadas as suas características e aplicações utilizadas.

Com base nas definições anteriores, é proposto como aplicar as técnicas de teste de software mais adequadas a cada arquitetura desenvolvimento de sistema.

1.2. Motivações

Na maioria das empresas constituídas da indústria nacional de software caracteriza-se por atividades de teste limitadas a uma única fase, abreviadas nos últimos estágios de desenvolvimento e executadas sem a alocação de recursos suficientes para a realização das mesmas.

Outra observação importante, é que na maioria das vezes muitos testes não seguem nenhuma padronização, ou seja, cada sistema ou mesmo cada grupo de trabalho dentro de uma mesma organização/sistema realizam os testes de forma distintas.

A situação tem causado, graves conseqüências, desde atrasos na entrega do software até um estouro no orçamento devido às correções que devem ser executadas.

Estudos realizados no E.U.A. pelas empresas IBM, GTE e TRW mostraram que corrigir um defeito após a fase de codificação custa 10 vezes mais do que corrigi-lo antes da fase de codificação [PERRY, 1995].

Em relação aos custos envolvidos na detecção e correção de defeitos, a situação agrava-se mais ainda quando as atividades de teste são abreviadas ou completamente excluídas do ciclo de vida de desenvolvimento de software ou, então, são realizadas de maneira equivocada, isto é, realizadas com o objetivo de mostrar que o software é isento de erros.

Dentro deste panorama cria-se um círculo vicioso no qual as atividades de teste não têm a sua importância reconhecida, no tocante à produção de softwares confiáveis a custo razoáveis, pois muitas empresas não têm consciência dos custos elevados envolvidos no processo de correção de erros após a implementação do software e vêem as atividades

de teste como impraticáveis – a respeito de tempo, custo e esforço envolvidos.

Apresentar os tipos de técnicas de teste - seus conceitos, seu funcionamento, seu processo de desenvolvimento e principalmente sua adequada aplicação nas diferentes arquiteturas de sistemas é a principal motivação para a elaboração deste trabalho.

1.3. Estrutura do Trabalho

Este trabalho está estruturado da seguinte forma:

- Capítulo 2 – Definições

Neste Capítulo são apresentadas as definições, a terminologia e as notações que serão utilizadas ao longo dos demais capítulos;

- Capítulo 3 – Histórico

Este capítulo apresenta a evolução da disciplina teste de software através do exame das mudanças experimentadas pelos modelos e processos de teste de software e do aumento do nível de profissionalismo com o qual os testes são tratados. A importância deste capítulo situa-se apenas como forma de apresentar o contexto da disciplina teste de software, mostrando como as atividades de teste de software evoluíram de um estado inicial – no qual o teste era a última fase do processo – até os dias atuais – onde o foco das atividades de teste moveram-se para seu início, distribuindo-se ao longo de todas as fases de desenvolvimento.

- Capítulo 4 – Técnicas de Teste de Software

Ao longo do capitulo as diferentes técnicas de teste de software são descritas, as quais serão utilizadas posteriormente:

- Caixa Branca
  - Teste de Caminho Básico
  - Teste de Condição
  - Teste de Fluxo de Dados
  - Teste de Laços

- Caixa Preta
  - Particionamento de Equivalência
  - Análise de Valor Limite
  - Teste de Fluxo de Controle
  - Teste de Fluxo de Transações

- Capítulo 5 – Arquiteturas de Sistemas

Este capítulo apresenta os conceitos e diferenças entre os tipos de arquiteturas de software:

- Tempo-Real
- Centrados em Bancos de Dados
- Cliente-Servidor

- Capítulo 6 – Aplicações de Técnicas de Testes

Neste capítulo, é abordado o principal assunto do trabalho, onde, reunindo as informações apresentadas até então como base, é

sugerido a técnica de teste seria mais adequado na aplicação de determinada arquitetura de sistema.

- Capítulo 7 – Conclusão

Este capítulo destaca as contribuições apresentadas pela realização da aplicação, os objetivos alcançados e as principais dificuldades encontradas em sua aplicação com o objetivo de propor trabalhos futuros.

- Capítulo 8 – Referências Bibliográficas

Este capítulo apresenta as referências bibliográficas utilizadas no desenvolvimento deste trabalho.

2. DEFINIÇÕES

2.1. Definições de Teste de Software

Segundo a definição do Dicionário Aurélio:

Teste [Do ingl. *test.*] ***S.m.*** **1.** Exame, verificação ou prova para determinar a qualidade, a natureza ou o comportamento de alguma coisa, ou de um sistema sob certas condições. **2.** Método, processo, procedimento ou meios utilizados para tal exame, verificação ou prova. **3.** *Psicol.* Medida ou cálculo de determinadas características afetivas, intelectuais (nível mental, aptidões, conhecimentos), sensoriais ou motoras de um indivíduo, que permite situá-lo objetivamente em relação a outros membros do grupo social a que ele pertence. **4.** *Brás. P. ext.* Prova, exame, verificação.

Software [Ingl., voc. Cunhado por analogia com *hardware*(q.v.), de *soft*, 'macio', 'mole' + *ware*, 'artigo', 'utensílio'] ***S.m.*** *Inform.* **1.** Em um sistema computacional, o conjunto dos componentes que não fazem parte do equipamento físico propriamente dito e que incluem as instruções e programas (e os dados a eles associados) empregados durante a utilização do sistema. **2.** Qualquer programa ou conjunto de programas de computador. **3.** *P. ext.* Produto que oferece um conjunto de programas e dados para uso em computador.

Da junção dos verbetes obteríamos que teste de software seria um "Método, processo ou procedimento para se realizar um exame, verificação ou prova para determinar a qualidade ou o comportamento de um conjunto de componentes ou programas de um sistema computacional".

Segundo alguns autores a definição de teste de software pode ser apresentada como:

- O teste é um processo da aquisição de confiança no fato de que um programa ou sistema faz o que se espera dele. [HETZEL, 1973]
- Teste é o processo de executar um programa ou sistema com a finalidade de encontrar erros. [MYERS, 1979]
- O processo de se experimentar ou avaliar um sistema por meios manuais ou automáticos, de modo a verificar se ele atende às necessidades especificadas ou a identificar as diferenças entre os resultados esperados e reais. [IEEE, 1983]
- Teste é qualquer atividade que vise a avaliar uma característica ou recurso de um programa ou sistema. Teste é a medida da qualidade de Software [HETZEL, 1983]

2.2. Terminologia

O Glossário de Software do IEEE define erro e falha da seguinte forma:

- Erro: Uma discrepância conceitual ou sintática que resulta em uma falha ou mais falhas em um software.
- Falha: Uma manifestação específica de um erro. Uma discrepância em um software que pode prejudicar sua capacidade de funcionar conforme pretendido. Um erro pode ser a causa de várias falhas.

2.3. Notações

Para a representação do fluxo de controle de um programa, utiliza-se uma notação denominada grafo de fluxo (ou grafo de programa) [BEIZER, 1983], [HOWDEN, 1981]. O grafo de fluxo descreve o fluxo de controle lógico usando a notação mostrada na figura 2.1. Cada instrução estruturada tem um símbolo de grafo correspondente.

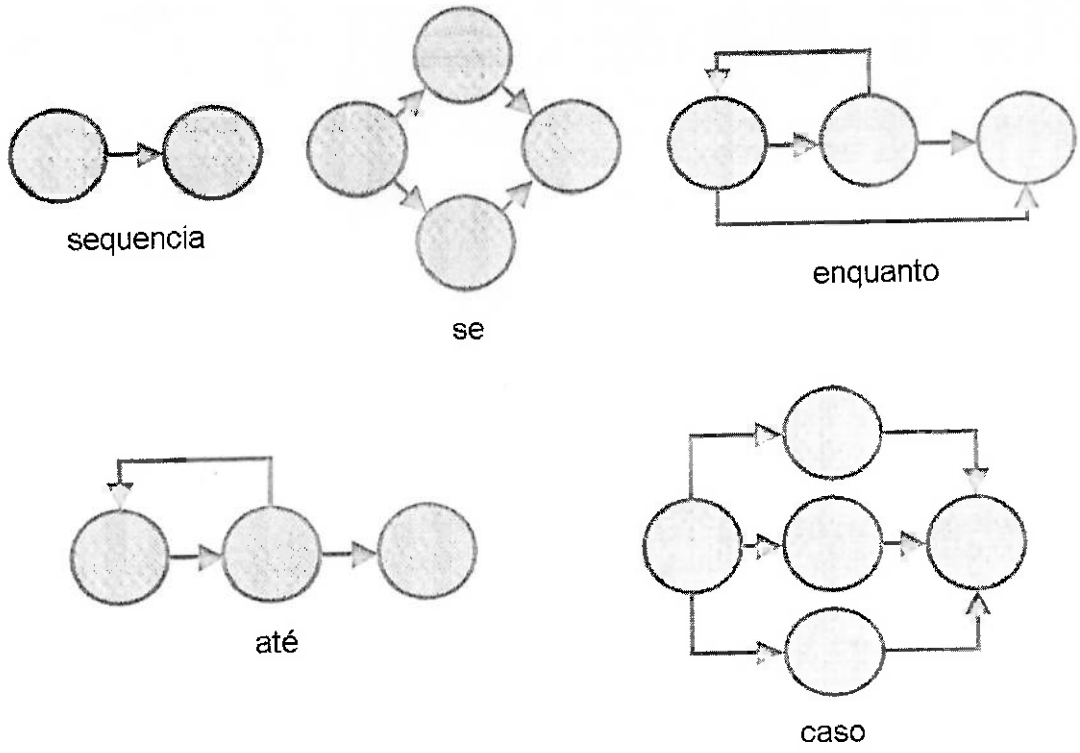


Figura 2-1 – Notação de grafo de fluxo [BEIZER, 1983]

Como exemplo de aplicação da utilização de um grafo de fluxo considere-se a representação de projeto procedimental – processo de transformação dos componentes estruturais de um software numa descrição procedimental do software – da figura 2.2, onde um fluxograma é utilizado para descrever a estrutura de controle do programa. A figura 2.3 apresenta o grafo de fluxo correspondente ao fluxograma da figura 2.2. Em relação, à figura 2.3, cada círculo, denominado nó de grafo de fluxo, representa uma ou

mais instruções. Uma seqüência de quadros de processamento e um losango de decisão podem levar a um único nó. Os arcos do grafo de fluxo, denominados ramos de grafo de fluxo, representam o fluxo de controle e são análogos aos arcos do fluxograma. Um ramo deve terminar em um nó, mesmo que o nó não represente quaisquer instruções. As áreas delimitadas pelos ramos e nós são denominadas regiões. Quando se contam as regiões, inclui-se a área fora do grafo como uma região. Cada nó que contém uma condição lógica é denominado nó predicativo e é caracterizado por dois ou mais ramos que saem dele.

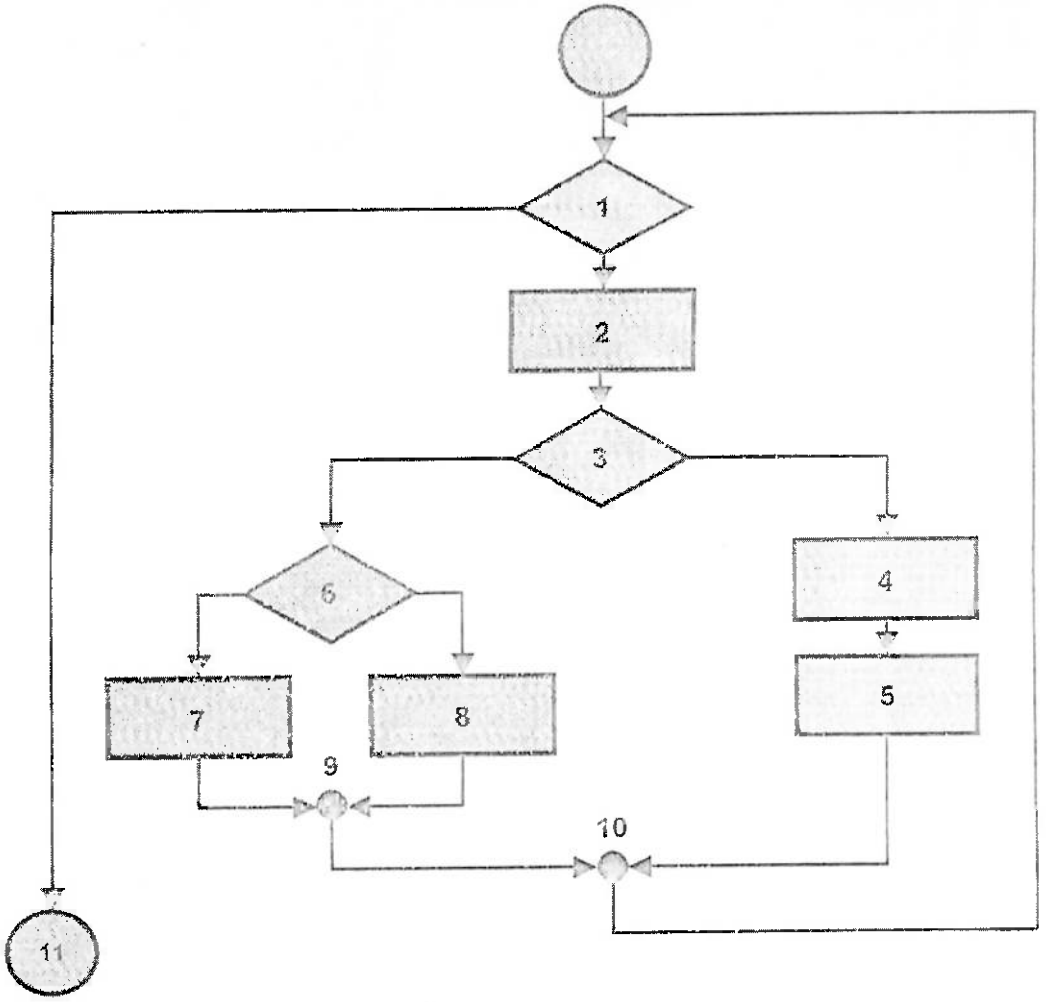


Figura 2-2 – Fluxograma [PRESSMAN, 1992]

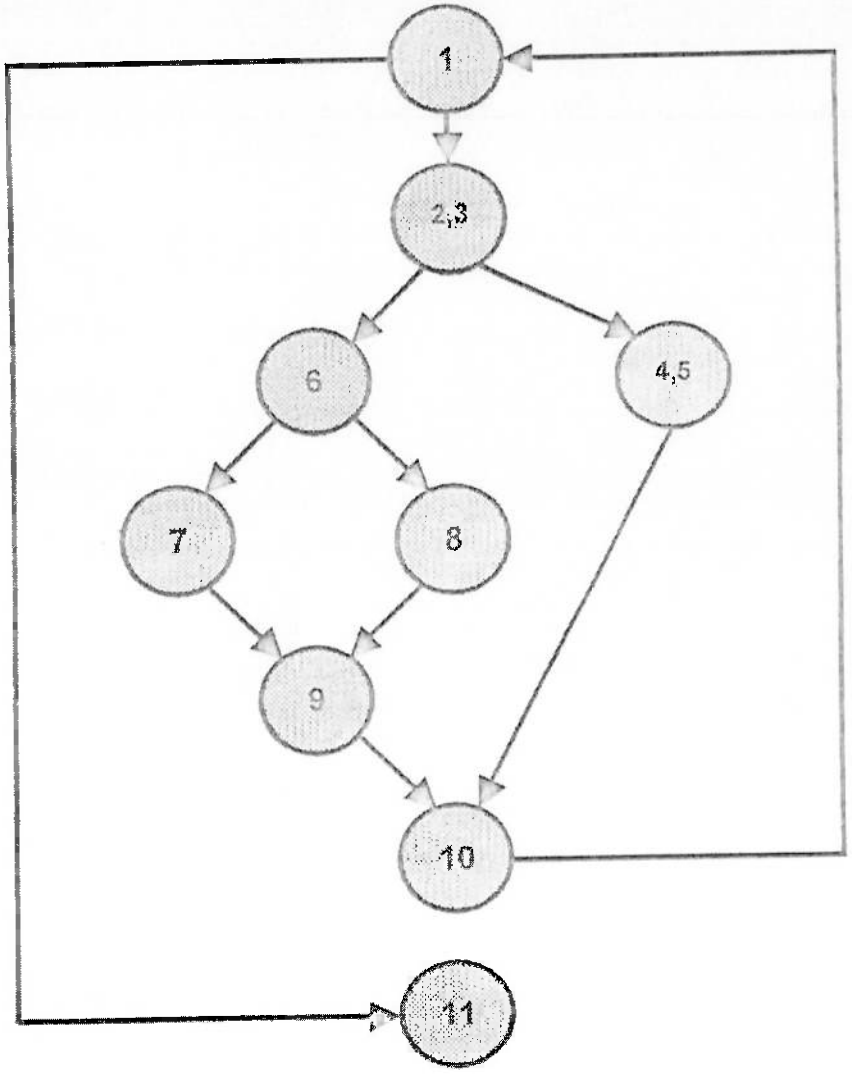


Figura 2-3 – Grafo de fluxo [PRESSMAN, 1992]

2.4. Estratégia de Testes

Para se desenvolver uma atividade de teste de maneira a atingir o objetivo de qualidade que se espera, não existe um teste único a ser realizado e sim uma combinação de vários testes ao longo do projeto. A estratégia descrita por Pressman (1997), prevê as seguintes etapas:

- **Teste de Unidade**: inicialmente os testes focalizam cada módulo individualmente, garantindo que ele funcione adequadamente como uma unidade;
- **Teste de Integração**: visa a construir o programa a partir dos módulos testados no nível de unidade, realizando-se ao mesmo tempo, testes para descobrir defeitos associados a interfaces entre os módulos;
- **Teste de Validação**: tem por objetivo testar o software depois de integrado e validar os requisitos funcionais e de desempenho estabelecidos durante a fase de análise;
- **Teste de Sistema**: testa o programa com outros elementos: hardware, pessoas, banco de dados.

Segundo Beizer (1984) são propostos alguns tipos de testes válidos: teste de recuperação ou tolerância a falhas, teste de segurança ou invasão, teste de estresse e teste de desempenho.

3. HISTÓRICO

A palavra teste é derivada do latim (testum), significando originalmente um pote de barro onde metais eram decompostos para a determinação da presença de diversos elementos ou medição de seu peso.

A notação específica de teste de programas surgiu quase ao mesmo tempo que os primeiros programas. Os programas executados nos primórdios da computação também tinham que ser testados e encontram-se menções ao teste de programas na literatura de processamento de dados já em 1950.

Durante a primeira metade de 1950, os testes eram considerados uma atividade secundária, englobando os esforços para a detecção de erros e para sua correção e eliminação. Só em 1957 o teste de software foi nitidamente separado da depuração.

A partir do fim da década de 1950, o teste de software vem assumindo uma importância crescente em decorrência de aspectos técnicos e econômicos. Os sistemas de computador eram falhos e deficientes, o custo e o impacto causados pela recuperação desses problemas tornavam-se cada vez maiores. Passou-se a dar ênfase a testes mais criteriosos por parte dos usuários e gerentes de desenvolvimento, e os tempos e recursos dedicados ao teste cresceram de maneira acentuada.

Em 1972 foi realizada na Universidade da Carolina do Norte a primeira conferência formal voltada ao teste de software, como subproduto dessa conferência foi definido que o termo testar abrange um grande número de atividades, todas elas associadas à aquisição de confiança no fato de que um programa ou sistema tem o desempenho esperado.

Desde então, muitas outras conferências e workshops foram realizados para tratar de qualidade, confiabilidade e engenharia de software, e pouco a pouco o teste de software tornou-se uma disciplina organizada dentro da informática.

O tema vem crescendo em importância com a necessidade de todos os setores da informática no sentido de criar métodos práticos e unanimidade quanto a definição de testes a fim de assegurar a qualidade dos produtos finais.

3.1. Modelos de Teste de Software

Ao longo da história, destacam-se quatro modelos de teste de software:

Modelo	Objetivo	Autor
Demonstração	Mostrar que o software satisfaz suas especificações	White, 1970
Destruição	Executar o software com a intenção de descobrir erros	Myers, 1979
Avaliação	Detectar erros no conjunto de requisitos, no projeto e na implementação do software	Adrion, 1982
Prevenção	Prevenir erros no conjunto de requisitos, no projeto e na	Gelperin, 1988

	implementação do software	

Tabela 3-1. - Principais modelos de teste de Software [ROTA, 2001]

Os modelos de teste de software apresentados na Tabela 3.1 são, basicamente, diferenciados pela extensão de suas atividades e por seus objetivos básicos. Os dois primeiros modelos – demonstração e destruição – são ditos modelos de fase e os dois últimos modelos – avaliação e prevenção – são ditos modelos do ciclo de vida [GELPERIN, 1988]

Os modelos de demonstração e destruição tem suas atividades focadas na execução do software, uma vez que, através desta execução, seus objetivos básicos são atingidos. Embora ambos modelos reconheçam que seja necessário planejar os testes e selecionar os dados a serem utilizados, nitidamente eles associam a palavra teste com execução de teste, sendo que estes termos acabam se tornando sinônimos. Desta associação advém a visão das atividades de teste como um estágio independente do ciclo de vida de desenvolvimento de software.

Já os modelos de avaliação e prevenção movem o foco das atividades de teste do final do projeto para seu início. Este movimento resulta na visão das atividades de teste não devem se limitar apenas aos últimos estágios de desenvolvimento, mas sim, realizadas ao longo de todos os estágios de desenvolvimento. Assim sendo, as atividades de teste são intimamente relacionadas às atividades de teste – planejamento dos testes a serem realizados, projeto dos casos de teste, execução destes casos de teste, etc – são associadas às atividades de análise (análise de fluxo de dados, provas de correção, etc.) e às atividades de revisão (revisões técnicas formais, inspeções e revisões) constituindo um amplo processo chamado ciclo de vida de teste.

Ao longo dos anos, da mesma forma que os sistemas computadorizados evoluíram, os testes de software também ficaram mais rigorosos na verificação do ciclo de vida do sistema.

O modelo demonstração diz que o objetivo dos testes é mostrar que o software satisfaz suas especificações. Se um determinado software executa com sucesso um determinado conjunto de testes, diz-se que este software satisfaz suas especificações. O modelo demonstração não oferece nenhuma diretriz de como se obter tal conjunto de testes e, portanto, um conjunto de testes inadequado pode mascarar um software deficiente.

O modelo destruição apresenta uma mudança drástica de ponto de vista. Para este modelo, um teste bem-sucedido é aquele capaz de descobrir sistematicamente diferentes classes de erros. Contudo, tal mudança de mentalidade introduz uma nova dificuldade: saber quando os testes devem cessar. Como não há como saber se, após a execução dos testes, o software ainda apresenta erros ou não, não há uma resposta satisfatória para responder esta questão.

Segundo Myers (1979) dois critérios são freqüentemente utilizados:

- Os testes devem cessar se o orçamento se esgotar;
- Os testes devem cessar se todos os casos de teste tiverem sido executados e nenhum erro tiver sido detectado.

O objetivo principal do modelo avaliação é a detecção e a remoção de erros o mais cedo possível, através de adição de atividades de validação e verificação às atividades de teste, incorporando-as ao longo de todas as fases do processo de desenvolvimento de software.

Já o modelo de prevenção foca suas atividades na prevenção de erros, aperfeiçoando o trabalho desenvolvido, principalmente, durante as

fases de levantamento de requisitos do projeto, através de revisões dos requisitos que o software deve atender – verificando se critérios de teste satisfatórios para todos os requisitos especificados podem ser definidos – e através de análises das opções de projeto selecionadas – identificando condições de teste que irão submeter à prova tais opções.

3.2. Evolução dos Modelos de Teste de Software

No inicio do desenvolvimento de sistemas, em meados de 1950, os testes de software basicamente eram realizados no sentido de verificar erros (depuração) da codificação executada anteriormente.

Os critérios utilizados para a seleção dos casos de teste eram completamente *ad hoc*, sendo baseados única e exclusivamente na experiência dos programadores e em seu conhecimento sobre o sistema em desenvolvimento [GELPERIN, 1988].

No período de inicio dos anos 60 até o fim da década de 70 – modelo de demonstração - as aplicações baseadas em computador passam a aumentar em quantidade, complexidade e custo, com isso, as atividades de teste de software passam a receber maior atenção, iniciando-se um processo de aperfeiçoamento da eficiência dos testes, no sentido de que estes sejam capazes de detectar erros antes que o software seja colocado em ambiente de produção.

Neste período também é caracterizada uma distinção entre depuração e teste:

- Depuração: Consiste em assegurar que o sistema pode ser executado;

- Teste: Consiste em assegurar que o sistema realiza as tarefas para as quais ele foi projetado.

A partir da publicação do livro de Myers em 1979 – "*The Art of Software Testing*" – o qual descreve o teste de software como sendo o processo de executar um programa com o objetivo de se encontrar erros, dá se inicio a utilização do modelo de destruição de teste de software.

Neste período as palavras depuração e teste ganham novos significados:

- Teste: Relaciona-se ao processo de revelar a presença de erros em um sistema;
- Depuração: Relaciona-se ao processo de localizar e corrigir esses erros.

Nestes dois últimos períodos a fase de teste é separada do processo de desenvolvimento de software, sendo colocada em um último estágio [GELPERIN, 1988].

Com a publicação do livro "*Guideline for Lifecycle Validation, Verification, and Testing of Computer Software*" do Instituto de Ciências da Computação e Tecnologia do Departamento Nacional de Padronizações (E.U.A.) e com a percepção de que quanto antes um erro é detectado, menos custoso será sua correção, dá se inicio a utilização do modelo de avaliação (1983 – 1987).

Para cada estágio do ciclo de vida de desenvolvimento de software são aplicados os conceitos de requisitos e produtos. O objetivo básico de cada uma das fases de avaliação é determinar quão bem os produtos atendem os requisitos.

O modelo de prevenção passa a ser utilizado a partir dos trabalho de Hetzel e Gelperin em 1988, trabalhos estes que generalizam métodos para testes de unidade – teste de componentes individuais de software, onde se considera apenas o domínio de entrada para a unidade sob teste e ignoram-se as demais unidades do sistema – e desenvolvem um método prático para gerenciamento de teste de software.

A idéia é prevenir erros ao longo de cada fase do ciclo de vida de desenvolvimento utilizando-se técnicas de teste e outras técnicas de avaliação à medida que o desenvolvimento avança – diferentemente do período do modelo de avaliação, onde tais técnicas eram utilizadas apenas no término de cada fase do ciclo de vida de desenvolvimento. Os critérios para a seleção de casos de teste são direcionados para atividades propensas a erros.

Os critérios para a seleção de casos de teste devem contemplar os seguintes pontos:

- Os casos de teste devem ser capazes de revelar erros. Estes casos de teste irão gerar resultados contraditórios às especificações se erros estiverem presentes [ROTA, 2001];
- Os casos de teste devem se integrados aos estágios do ciclo de vida de desenvolvimento, pois assim, cada estágio gera os seus próprios casos de teste. O esforço necessário para gerar casos de teste durante cada estágio do ciclo de vida de desenvolvimento é menor que o esforço necessário para produzir uma grande quantidade de casos de teste de igual eficiência em um estágio final, à parte do ciclo de vida de desenvolvimento [ROTA, 2001].

3.3. Situação Atual

Atualmente, a situação encontrada na maioria das empresas é a seguinte: os testes são mal estruturados e feitos de forma fortemente individualizada; poucos planejamentos de projeto prevêem recursos para a realização de testes; não há um plano de execução dos testes que estabeleça o quê testar e nem quais resultados se espera obter; os testes não são conduzidos de maneira sistemática e organizada através de um método bem definido e planejado; e não se mantém registros do tempo gasto e dos recursos utilizados para a realização dos testes, para que se possa calcular o custo do mesmo. Ainda hoje, muitos profissionais da área de desenvolvimento de software pensam que os softwares são somente escritos e depois testados e depurados.

4. TÉCNICAS DE TESTE DE SOFTWARE

4.1. Teste de Caixa Branca

Os testes de caixa branca referem-se a um minucioso exame dos detalhes internos de um software. Usando-se métodos de teste de caixa branca pode-se derivar casos de teste que assegurem que todos os caminhos de execução através do software sejam executadas pelo menos uma vez, que exercitem todas as condições lógicas para valores verdadeiros e falsos, que exercitem as estruturas de dados internas e que executem todos os laços [PRESSMAN, 2001].

Pode-se se citar as técnicas de caixa-branca mais utilizadas:

- Teste de Caminho Básico
- Teste de Condição
- Teste de Fluxo de Dados
- Teste de Laços

4.1.1. Teste de Caminho Básico

O teste de caminho básico é uma técnica de teste de caixa branca proposta por Tom McCabe. O método possibilita medir a complexidade lógica de um programa e utilizar esta medida como referência para definir um conjunto básico de caminhos de execução. Os casos de teste projetados para exercitarem este conjunto básico de caminhos de execução garantem

que todas as instruções do programa sejam executadas pelo menos uma vez [BEIZER, 1984].

A complexidade ciclomática é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usado no contexto do método de teste de caminho básico, o valor da complexidade ciclomática define o número de caminhos independentes do conjunto total de caminhos de execução de um programa e fornece um limite máximo para o número de testes que devem ser realizados para garantir que todas as instruções sejam executadas pelo menos uma vez [PRESSMAN, 2001].

Um caminho independente é qualquer caminho através do programa que introduza pelo menos um novo conjunto de instruções ou uma nova condição. Em termos de grafo de fluxo (Item 2.3), um caminho independente deve incluir pelo menos um ramo que não tenha sido atravessado antes. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo da figura 4.1. é:

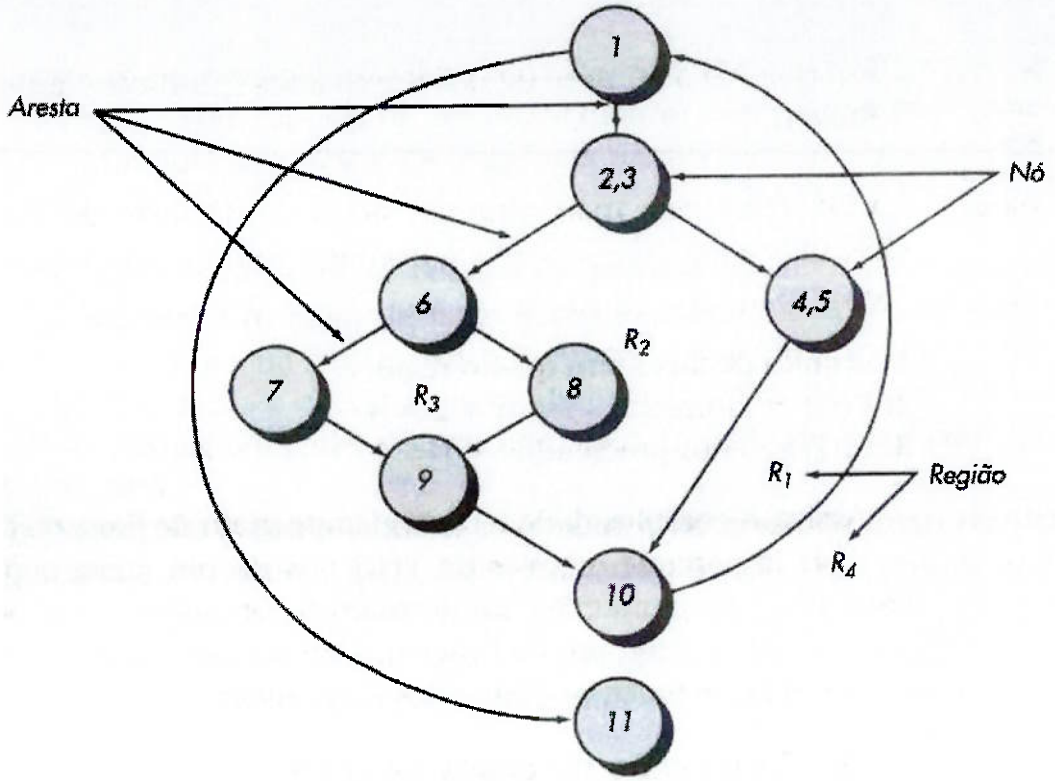


Figura 4-1 – Grafo de fluxo [PRESSMAN, 2001]

Caminho 1: 1 – 11

Caminho 2: 1 – 2 – 3 – 4 – 5 – 10 – 1 – 11

Caminho 3: 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11

Caminho 4: 1 – 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11

Pode-se notar que cada caminho introduz um novo ramo. O caminho 1 – 2 – 3 – 4 – 5 – 10 – 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11 não é considerado um caminho independente porque ele é simplesmente uma combinação de caminhos já especificados e não atravessa nenhum ramo novo.

Os caminhos 1, 2, 3 e 4 compreendem um conjunto básico de caminhos de execução para o grafo de fluxo mostrado na figura 4.1. Ou seja, se for possível projetar casos de teste para garantir a execução destes caminhos, cada instrução terá a garantia de ter sido executada pelo menos

uma vez e cada condição lógica terá sido executada com resultado verdadeiro e falso. Observe que o conjunto básico de caminhos de execução não é único. Diferentes conjuntos básicos de caminhos de execução podem ser derivados para um determinado programa.

O valor da complexidade ciclomática indica quantos caminhos independentes devem ser procurados.

A complexidade ciclomática tem suas bases na teoria dos grafos e pode ser calculada numa das três seguintes formas:

1. O número de regiões do grafo de fluxo corresponde à complexidade ciclomática;

2. A complexidade ciclomática, para um grafo de fluxo é definida como :

$$M = L - N + (2 \times P)$$

Onde,

M = Complexidade ciclomática
L = Número de ramos do grafo de fluxo
N = Número de nós do grafo de fluxo
P = Número de partes desconexas do grafo (por exemplo, um programa principal e uma sub-rotina)

3. A complexidade ciclomática, para um grafo de fluxo é definida como :

$$M = R + 1$$

Onde,

M = Complexidade ciclomática
R = Número de nós predicativos contidos no grafo de fluxo

Referenciando-se a figura 4.1., a complexidade ciclomática pode ser obtida usando qualquer uma das três formas apresentadas:

1. O grafo de fluxo contém 4 regiões;
2. L = 11 Ramos, N = 9 Nós, P = 1 parte desconexa, daí M = 11 – 9 + (2 x 1) = 4 ;
3. R = 3 Nós predicativos, daí M = 3 + 1 = 4.

Portanto, a complexidade do grafo de fluxo é 4.

Desta forma, o valor de M fornece um limite máximo para o número de caminhos independentes que constitui o conjunto básico de caminhos de execução e, por conseguinte, um limite máximo do número de casos de teste que devem ser projetados e executados para garantir a cobertura de todas as instruções e de todas as condições lógicas de um determinado programa.

4.1.2. Teste de Condição

O teste de condição é um método de projeto de casos de teste que se concentra em testar cada condição lógica contida num módulo de programa [TAI, 1989].

Uma condição simples é uma variável booleana ou uma expressão relacional que assume a seguinte forma:

E_1 < operador relacional > E_2

Onde E_1 e E_2 são expressões aritméticas e < operador relacional > é um dos seguintes operadores : <, ≤, =, ≠, > ou ≥. Uma condição composta é constituída de duas ou mais condições simples, operadores booleanos e parênteses. Os operadores booleanos permitidos numa condição composta são: OR, AND e NOT. Uma condição sem expressões relacionais é chamada expressão booleana.

Portanto, os possíveis tipos de componentes de uma condição incluem um operador booleano, uma variável booleana, uma par de parênteses booleano (ao redor de uma condição simples ou composta), um operador relacional ou uma expressão aritmética.

Se uma condição estiver incorreta, então pelo menos um componente da condição esta incorreto. Deste modo, entre os tipos de erro de uma condição incluem-se os seguintes:

- Erro de operador booleano (a existência de operadores booleanos incorretos / faltando / extras)
- Erro de variável booleana
- Erro de parênteses booleanos
- Erro de operador relacional
- Erro de expressão aritmética

Os métodos de teste de condição apresentam duas vantagens. Primeiro, a medição da cobertura do teste de uma condição é simples. Segundo, a cobertura de teste das condições de um programa fornece orientação para a geração de testes adicionais para o programa.

O objetivo do teste de condição é detectar não somente erros nas condições de um programa, mas também outros erros no programa – implementações para tratamentos de condições de erros incorretas, trechos de código inacessíveis, respostas inadequadas, etc. Se um conjunto de testes para um programa for eficiente para detectar erros nas condições contidas neste programa, é provável que este conjunto de testes também seja eficiente para detectar outros tipos de erros existentes neste programa.

Uma série de métodos de teste de condição tem sido proposta, as quais, podemos destacar o teste de ramos, o teste de domínio e o teste de operadores booleanos e relacionais [TAI, 1989].

4.1.2.1. Teste de Ramos

Proposto por Myers (1979), o teste de ramos é o método de teste de condição mais simples. Dada uma condição C composta, os ramos verdadeiro e falso de C, todas as condições simples em C precisam ser executadas pelo menos uma vez.

4.1.2.2. Teste de Domínio

Proposto por White (1980), o teste de domínio requer quer três testes sejam derivados para um expressão relacional:

$$E_1 < \text{operador relacional} > E_2$$

Três testes são exigidos para que o valor de E_1 seja menor, igual ou maior que o valor de E_2 . Se < operador relacional > estiver incorreto e E_1 e E_2 estiverem corretos, então estes três testes garantem a detecção do erro

de operador relacional. Para detectar erros em E_1 e E_2, um teste que faça o valor de E_1 maior ou menor que o valor de E_2 deve fazer a diferença entre estes dois valores tão pequena quanto possível[HOWDEN, 1982].

Para uma expressão booleana com n variáveis, todos os 2^n testes possíveis são exigidos (n > 0). Este método pode detectar erros de operadores, variáveis e parênteses booleanos. Na prática, esta estratégia é aplicável somente se n for pequeno.

4.1.2.3. Teste de Operadores Booleanos e Relacionais (BRO, do inglês Boolean and Relational Operator testing)

Proposto por Tai (1993), o BRO é um método de teste de condição, que garante a detecção de erros de operadores relacionais e ramificações numa condição, contanto que todas as variáveis booleanas e operadores relacionais da condição ocorram apenas uma vez e não tenham variáveis comuns.

O método BRO usa restrições de condição para condição C. Uma restrição de condição para C com n condições simples é definida como (D_1, D_2, ..., D_n), onde D_i ($1 < i \leq n$) é um símbolo que especifica uma restrição ao resultado da i-ésima condição simples da condição C. Diz-se que uma restrição de condição D para a condição C é coberta por uma execução de C se durante esta execução de C o resultado de cada condição simples em C satisfizer a correspondente restrição em D.

Para uma variável booleana B, especifica-se uma restrição ao resultado de B que estabeleça que o valor de B deva ser verdadeiro (TRUE) ou falso (FALSE). Similarmente, para uma expressão relacional, os símbolos

>, = e < são usados para especificar restrições ao resultado da expressão relacional.

Para melhor compreensão, seguem três exemplos:

Exemplo 1

Considere-se a seguinte condição:

$$C_1: B_1 \& B_2$$

Onde B_1 e B_2 são variáveis booleanas. A restrição de condição C_1 tem a forma (D_1, D_2), onde o valor de D_1 e D_2 são verdadeiro (TRUE) ou falso (FALSE). O valor (TRUE, FALSE) é uma restrição de condição para C_1 e é coberto pelo teste que faz o valor de B_1 ser verdadeiro e o valor de B_2 ser falso. O método de teste BRO exige que o conjunto de restrições (TRUE, TRUE), (FALSE, TRUE), (TRUE,FALSE) seja coberto pelas execuções de C_1. Se C_1 estiver incorreta devido a um ou mais erros de operadores booleanos, pelo menos um par de conjuntos de restrições detectará o erro.

Exemplo 2

Considere-se uma condição que tenha a forma

$$C_2: B_1 \& (E_3 = E_4)$$

Onde B_1 seja uma expressão booleana e E_3 e E_4 sejam expressões aritméticas. Uma restrição de condição para C_2 tem a forma (D_1, D_2), onde o valor de D_1 é verdeiro (TRUE) ou falso (FALSE) e o valor de D_2 é >, = ou <. Como a condição C_2 possui a mesma forma que a condição C_1 (Exemplo 1), exceto que a segunda condição simples em C_2 é uma expressão relacional, pode-se construir um conjunto de restrições para C_2 ao modificar-se o

conjunto de restrições (TRUE, TRUE), (FALSE, TRUE), (TRUE, FALSE) definido para C_1. Note que TRUE para ($E_3 = E_4$) implica = e que FALSE para ($E_3 = E_4$) implica < ou >. Ao substituir-se (TRUE, FALSE) por (TRUE, <) e (TRUE, >), o conjunto resultante de restrições para C_2 será (TRUE, =), (FALSE, =), (TRUE, <), (TRUE,>). A cobertura do conjunto resultante garantirá a detecção de erros de operadores relacionais e booleanos em C_2.

Exemplo 3

Considere-se a condição que tenha a seguinte forma

$$C_3: (E_1 > E_2) \& (E_3 = E_4)$$

Onde E_1, E_2, E_3 e E_4 sejam expressões aritméticas. Uma restrição de condição para C_3 tem a forma (D_1, D_2), onde o valor de D_1 e D_2 são >, = ou <. Como a condição C_3 é uma expressão relacional, pode-se construir um conjunto de restrições para C_3 ao modificar-se o conjunto de restrições para C_2(Exemplo 2), obtendo-se:

(>, =), (=, =), (<, =), (>, <), (>, >)

A cobertura do conjunto de restrições acima garantirá a detecção de erros de operadores relacionais em C_3.

4.1.3. Teste de Fluxo de Dados

A técnica de teste de fluxo de dados seleciona caminhos de teste de um programa de acordo com as localizações das definições e usos de variáveis no programa [FRANK, 1993] e [WEISS,1993].

Para exemplificar a abordagem dos testes de fluxo de dados, supor que cada instrução de um programa seja atribuída um número de instrução único e que cada função não modifique seus parâmetros ou variáveis globais.

Para uma instrução, cujo número de instrução seja S, temos:

DEF(S) = {X} – Instrução S contém uma definição de X
USE(S) = {X} – Instrução S contém um uso de X

Se a instrução S for um Se (If) ou um Laço (Loop), seu conjunto DEF deverá ser vazio e seu conjunto USE baseia-se na condição da instrução S. Diz-se que a definição da variável X na instrução S é "viva" na instrução S' se aí existir um caminho da instrução S para a instrução S' o qual não contenha nenhuma outra definição de X.

Uma associação definição-uso (DU) da variavel X tem a forma (X, S, S'), onde S e S' são números de instrução, X esta em DEF(S) e USE(S') e a definição de X na instrução S é viva na instrução S'.

Uma estratégia de teste de fluxo de dados simples consiste em exigir que cada associação DU seja coberta pelo menos uma vez. Refere-se a essa estratégia como estratégia de teste DU. Foi demonstrado que o teste DU não garante a cobertura de todos os ramos de um programa. Porém, um ramo não tem a garantia de ser coberto pelos testes DU somente em raras situações, como por exemplo, construções if-then-else em que parte then não tenha nenhuma definição de uma variável e a parte else não exista. Nessa situação, o ramo else da instrução if não é necessariamente coberto pelo teste DU [WEYUKER, 1985].

Uma vez que as instruções de um programa relacionam-se entre si de acordo com as definições e usos de variáveis, a abordagem do teste de fluxo de dados é eficiente para a proteção contra erros. Porém, os problemas de

medir a cobertura de teste e a seleção de caminhos de teste para o teste de fluxo de dados são mais complexos do que os correspondentes problemas para o teste de condição.

4.1.4. Teste de Laço

O teste de laço é uma técnica de teste de caixa branca que se concentra exclusivamente na validade das construções de laços. Quatro diferentes classes de laços podem ser definidas: laços simples, laços concatenados, laços alinhados e laços não estruturados[BEIZER, 1983]. A figura 4.2 exibe estas quatro classes de laços.

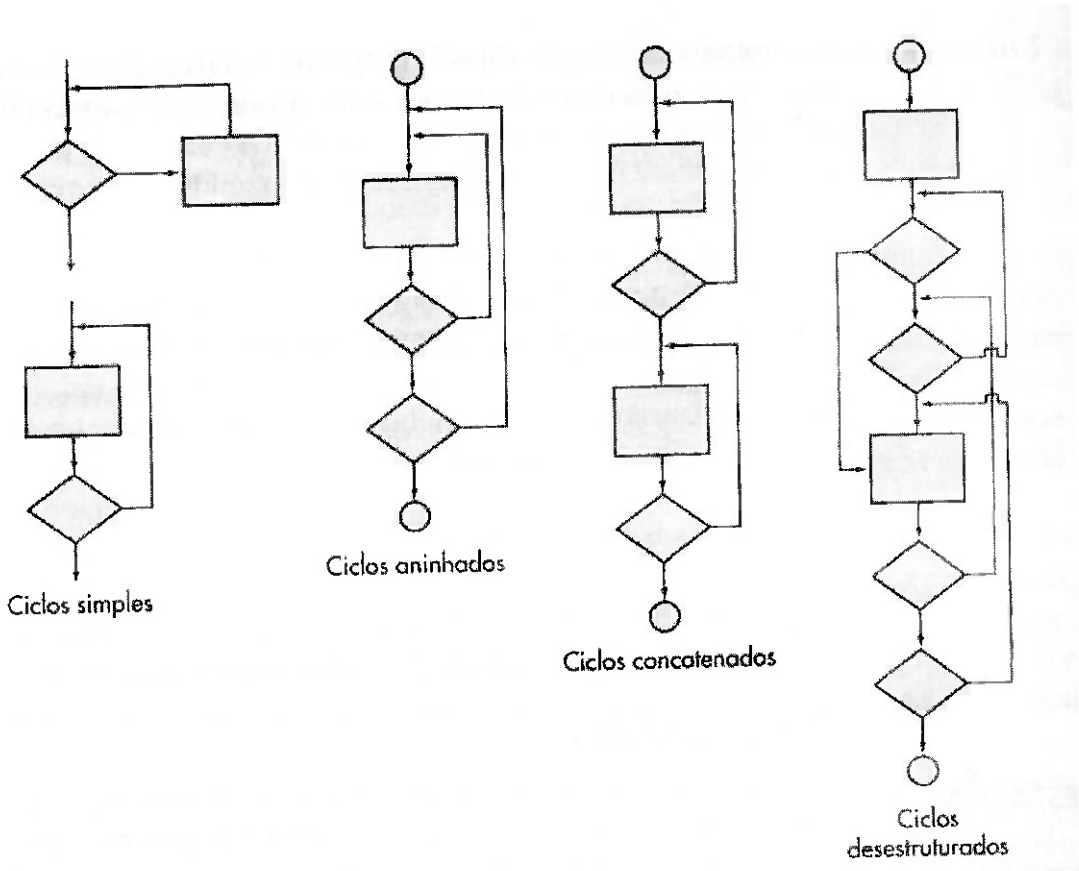


Figura 4-2 – Classes de laços [BEIZER, 1983]

4.1.4.1. Laços Simples

O seguinte conjunto de teste deve ser aplicado aos laços simples, onde n é o número máximo de passagens possíveis através do laço:

1. Pule o laço inteiramente;
2. Execute somente uma passagem através do laço;
3. Excute duas passagens através do laço, onde m < n;
4. Execute n – 1, n e n + 1 passagens através do laço.

4.1.4.2. Laços Concatenados

Os laços concatenados podem ser testados utilizando-se a abordagem definida para os laços simples se cada um dos laços for independente dos demais. Porém, se dois laços estiverem concatenados e o contador de laços para o laço 1 for usado como o valor inicial para o laço 2, então os laços não são independentes. Neste caso a abordagem aplicada para os laços aninhados é recomendada.

4.1.4.3. Laços Aninhados

Estendendo-se a abordagem aplicada aos laços simples para os laços aninhados, o número de testes possíveis cresce geometricamente à medida que a quantidade de laços aninhados se eleva. Isto resultaria num número de testes pouco prático. Por isso, Beizer (1993) sugere uma abordagem que ajudará a reduzir o número de testes:

1. Comece pelo laço localizado mais internamente. Fixando todos os outros laços com valores de parâmetro de iteração mínimos;
2. Realize testes de laços simples para o laço mais interno, mantendo ao mesmo tempo os laços externos em seus valores de parâmetro de iteração mínimos. Adicione outros testes para valores fora da faixa ou valores típicos;
3. Trabalhe de dentro para fora, realizando testes para o laço seguinte, mas mantendo todos os outros laços externos em valores de parâmetro de iteração mínimos e outros laços aninhados em valores típicos;
4. Continue até que todos os laços tenham sido testados.

4.1.4.4. Laços Não Estruturados

Sempre que possível esta classe de laços deve ser re-projetada para refletir o uso das construções de programação estruturada.

4.2. Teste de Caixa Preta

Segundo Myers (1979), as técnicas de teste de caixa preta concentram-se nos requisitos funcionais do software, procurando descobrir erros nas seguintes categorias:

1. Funções incorretas ou ausentes;
2. Erros de interface;
3. Erros nas estruturas de dados ou no acesso a estes;
4. Erros de desempenho;
5. Erros de iniciação e término

As principais técnicas de caixa preta podem ser citadas abaixo:

- Particionamento de equivalência
- Análise de Valor Limite
- Teste de Fluxo de Controle
- Teste de Fluxo de Transações

4.2.1. Particionamento de Equivalência

O particionamento de equivalência é uma técnica de teste de caixa preta que divide o domínio de entrada de um programa em classes de dados a partir das quais os casos de teste são projetados. Um caso de teste ideal é capaz de descobrir uma classe de erros, por exemplo, processamento incorreto de todos os dados do tipo caractere [MYERS, 1979].

O projeto de casos de teste para o particionamento de equivalência baseia-se numa avaliação de classes de equivalência para uma condição de entrada. Uma classe de equivalência representa um conjunto de estados validos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor numérico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. O particionamento de equivalência procura definir um caso de teste que descubra classes de erros, reduzindo assim o número total de casos de teste que devem ser desenvolvidos. Colocando de uma outra forma, se um caso de teste baseado em um elemento pertence à uma determinada classe de equivalência não detectar nenhum erro é improvável que um outro caso de teste baseado em um outro elemento pertencente à mesma classe de equivalência detecte algum erro.

As classes de equivalência podem ser definidas de acordo com as seguintes diretrizes:

1. Se uma condição de entrada especificar um intervalo, uma classe de equivalência valida e duas classes de equivalência invalidas são definidas. Por exemplo, se o mês do ano informado pelo usuário deve estar compreendido entre 1 e 12, aonde 1 ≤ mês ≤ 12 é uma classe de equivalência válida e mês < 1 ou mês > 12 é uma classe de equivalência invalida.
2. Se uma condição de entrada exigir um valor específico, uma classe de equivalência válida e duas classes de equivalência inválidas são definidas. Por exemplo, se um jogo possui, no máximo, dois jogadores, um dos dois jogadores é uma classe de equivalência válida e nenhum jogador ou mais de dois jogadores é uma classe de equivalência inválida.
3. Se uma condição de entrada especificar um elemento de um conjunto, uma classe de equivalência válida e uma classe de equivalência invalida são definidas. Por exemplo, se uma cor deve ser uma dos seguintes elementos: Verde, Amarelo, Azul e Branco. Qualquer um destes elementos é uma classe de equivalência válida e Vermelho é uma classe de equivalência invalida.
4. Se uma condição de entrada for um vetor booleano, um classe de equivalência válida e uma classe de equivalência inválida são definidas. Por exemplo, se o primeiro caractere de uma senha alfanumérica deve ser uma letra. Esta letra é uma classe de equivalência válida e um digito como primeiro caractere desta senha é uma classe de equivalência inválida.
5. Se há suspeita de que os elementos de uma classe de equivalência são tratados de maneira diferente por um programa, então esta classe de equivalência deve ser subdividida em classes de equivalência menores. Por exemplo,

se o valor do ingresso de um zoológico varia de acordo com a idade, de tal forma que se tem entrada franca para pessoas menores de 6 anos, R$ 5,00 para pessoas entre 6 e 12 anos e R$ 10,00 para pessoas maiores de 12 anos. De acordo com a técnica de teste de particionamento de equivalência, uma pessoa de 4 anos representa uma classe de equivalência para o domínio de entrada 0 ano ≤ idade ≤ 6 anos, uma pessoa de 10 anos representa uma classe de equivalência para o domínio de entrada 6 anos < idade ≤ 12 anos e uma pessoa com 16 anos representa uma classe de equivalência para o domínio de entrada idade > 12 anos.

4.2.2. Análise de Valor Limite

Segundo Myers (1979), a análise de valor de limite é uma técnica de teste que complementa o particionamento de equivalência.

Em vez de selecionar qualquer elemento de uma classe de equivalência, a análise de valor limite projeta casos de teste nas extremidades de uma classe, ou seja, se uma condição de entrada especificar um intervalo para os dados de entrada, casos de teste devem ser projetados com os valores mínimo e máximo e com valores imediatamente abaixo e imediatamente acima dos valores mínimo e máximo deste intervalo.

A análise do valor limite também projeta casos de teste para exercitar os dados de saída. Por exemplo, se um software fornece como resposta um valor compreendido entre 0 e 100, deve-se projetar casos de teste capazes de gerar como resposta 0 e 100, além disso, se possível, deve projetar casos de teste capazes de gerar repostas menores que 0 e maiores que 100.

4.2.3. Teste de Fluxo de Controle

O teste de fluxo de controle é uma técnica aplicada a maioria dos softwares, sendo referenciada como o modelo fundamental do teste de caixa preta, pois este tipo de teste é a base para as demais técnicas apresentadas [BEIZER, 1995].

O teste começa a partir da criação de um modelo de grafo de fluxo de controle com base nos documentos de requisitos e especificações. Também são muito utilizados a criação de listas com os procedimentos do sistema ou programa, porém grafos reduzidos são melhor aplicados à técnica.

Na construção dos grafos, os estados combinados devem ser evitados – se possível, deve-se, desmembrar em grafos menores – Porém, no caso de a utilização de estados combinados, o uso de uma tabela verdade é indicado. Por exemplo, o grafo de fluxo de controle (combinado) e sua tabela verdade são os das figuras 4.3. e 4.4.:

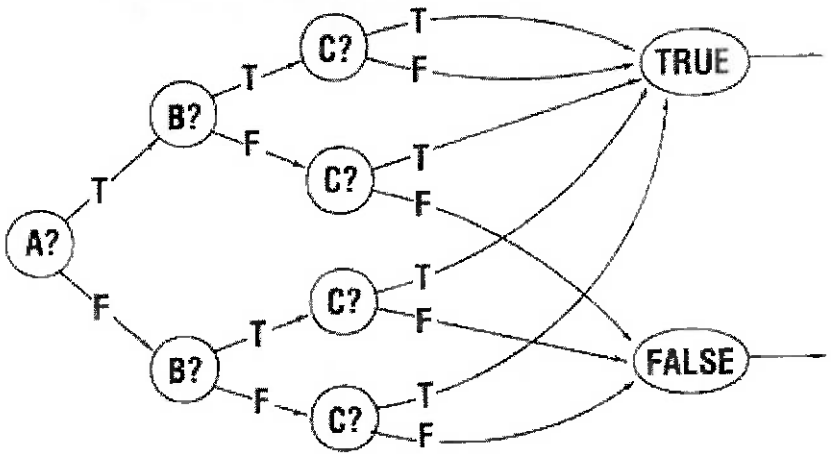


Figura 4-3 – Grafo de fluxo de controle [BEIZER, 1995]

A	B	C	A&B or C
TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	TRUE
TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE
FALSE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE

Tabela 4-1 - Tabela Verdade [BEIZER, 1995]

A partir de grafos reduzidos selecionam-se segmentos com inicio e fim com apenas um nó e define-se como os estados estão relacionados com este e com os outros segmentos. Com isso, os testes são definidos de acordo com os caminhos criados, eliminando os caminhos já percorridos.

Os resultados dos testes dos caminhos selecionados são interpretados de acordo com os estados (condições ou equações) e seus valores de entrada e saída, com isso soluções complexas devem ser revisadas para a criação, se possível, de um novo fluxo.

4.2.4. Teste de Fluxo de Transações

O teste de fluxo de transações é um modelo efetivo para testes de sistemas integrados. O modelo tem por objetivo inicial identificar as transações para serem modeladas, principalmente, quanto ao seu inicio e seu final de execução e as circunstâncias em que esta acontece.

Um modelo detalhado do processo deve ser construído afim de que seja possível acompanhar as filas e como as transações se interagem uma com as outras no sistema, tanto a nível isolado como no sistema inteiro.

5. ARQUITETURA DE SISTEMAS

Este capitulo tem por objetivo descrever, de forma resumida, as principais arquiteturas de sistemas utilizadas atualmente:

- Tempo-Real
- Centrados em Bancos de Dados
- Cliente-Servidor

5.1. Sistemas de Tempo-Real

Excetuando sistemas computacionais - normalmente identificados como "Sistemas Transformacionais" que calculam valores de saída a partir de valores de entrada e depois terminam seus processamentos como ocorre, por exemplo, com compiladores, programas de engenharia econômica e programas de cálculo numérico - uma grande parte dos sistemas computacionais atuais interagem permanentemente com os seus ambientes. Entre esses, distingue-se os chamados Sistemas Reativos que reagem enviando respostas continuamente aos estímulos de entrada vindos de seus ambientes. Sistemas de tempo-real de uma forma geral se encaixam neste conceito de sistemas reativos:

- Um Sistema de Tempo-real (STR) é um sistema computacional que deve reagir a estímulos oriundos do seu ambiente em prazos específicos.

O entendimento desses prazos resulta em requisitos de natureza temporal sobre o comportamento desses sistemas. Em conseqüência, em cada reação, o sistema de tempo-real deve entregar um resultado correto

dentro de um prazo específico, sob pena de ocorrer uma falha temporal. O comportamento correto de um sistema de tempo-real, portanto, não depende só da integridade dos resultados obtidos (correção lógica ou "*correctness*") mas também dos valores de tempo em que são produzidos (correção temporal ou "*timeliness*"). Uma reação que ocorra além do prazo especificado pode ser sem utilidade ou até representar uma ameaça. [STANKOVIC, 1990], [RAMAMRITHAN, 1990].

A maior parte das aplicações tempo-real se comportam então como sistemas reativos com restrições temporais. A reação dos sistemas de tempo-real aos estímulos vindos do ambiente externo ocorre em tempos compatíveis com as exigências do ambiente e mensuráveis na mesma escala de tempo. A concepção do sistema de tempo-real é diretamente relacionada com o ambiente no qual está relacionado e com o comportamento temporal do mesmo. Na classe de Sistema de Tempo-real na qual se encontram os sistemas embutidos ("*Embedded Systems*") e os sistemas de supervisão e controle, distingue-se entre o Sistema a Controlar, o Sistema Computacional de Controle e o Operador. O Sistema a Controlar e o Operador são considerados como o Ambiente do Sistema Computacional. A interação entre os mesmos ocorre através de interfaces de instrumentação (compostas de sensores e atuadores) e da interface do operador. A figura 5.1. representa esse tipo de Sistema de Tempo-real.

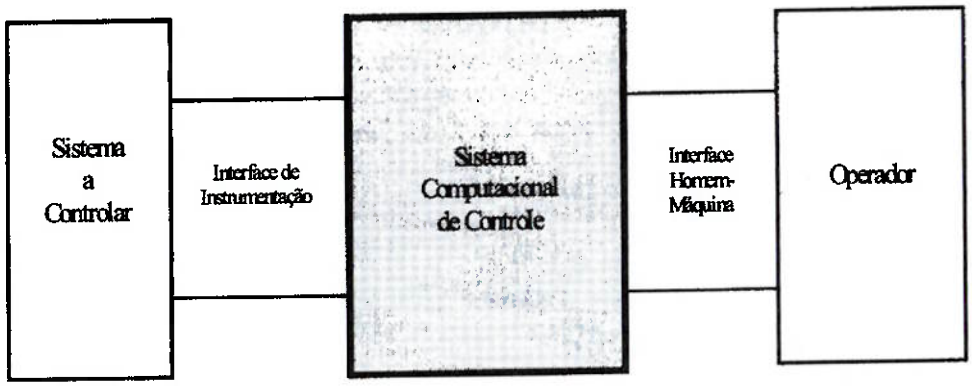


Figura 5-1 – Sistema tempo-real [RAMAMRITHAN, 1990]

Existem também situações nas quais as restrições temporais não são impostas pelo comportamento dinâmico de um eventual sistema a controlar mas pelas exigências dos serviços a serem oferecidos a um usuário humano ou computacional (p.ex. no caso do manuseio ou da apresentação de vídeos em sistemas multimídias). Nesses casos utiliza-se a noção de Serviço de Tempo-real como sendo um serviço que deve ser oferecido dentro de restrições de tempo impostas por exigências externas ao próprio Sistema Computacional [Delta-4, 1991]. Então, numa generalização podemos assumir que:

- Um Sistema de Tempo-real deve ser então capaz de oferecer garantias de correção temporal para o fornecimento de todos os seus serviços que apresentem restrições temporais.

5.2. Sistemas Centrados em Banco de Dados

Um Sistema Gerenciador de Banco de Dados (SGBD) é constituído por um conjunto de dados associados a um conjunto de programas para acesso a esses dados. O conjunto de dados, comumente chamado banco de dados. O principal objetivo de um SGBD é proporcionar um ambiente tanto conveniente quanto eficiente para a recuperação e armazenamento das informações do banco de dados.

Sistemas de banco de dados são projetados para gerir grandes volumes de informações. O gerenciamento de informações implica a definição das estruturas de armazenamento das informações e a definição dos mecanismos para a manipulação dessas informações. Ainda, um sistema de banco de dados deve garantir a segurança das informações armazenadas contra eventuais problemas com o sistema, além de impedir

tentativas de acesso não autorizadas. Se os dados são compartilhados por diversos usuários, o sistema deve evitar a ocorrência de resultados anômalos [SILBERSCHATZ, 1997], [KORTH, 1997], [SUDARSHAN, 1997].

A importância da informação na maioria das organizações - que estabelece o valor do banco de dados - tem determinado o desenvolvimento de um grande conjunto de conceitos e técnicas para a administração eficaz desses dados.

Um sistema de banco de dados está dividido em módulos específicos, de modo a atender a todas as funções do sistema. Algumas das funções do sistema de banco de dados podem ser fornecidas pelo sistema operacional. Na maioria das vezes, o sistema operacional do computador fornece apenas as funções essenciais, e o sistema de banco de dados deve ser construído nessa base. Assim, o projeto do banco de dados deve considerar a interface entre o sistema de banco de dados e o sistema operacional.

Os componentes funcionais do sistema de banco de dados podem ser divididos pelos componentes de processamento de consultas e pelos componentes de administração de memória. Os componentes de processamento de consultas incluem:

- **Compilador DML**: traduz comandos DML da linguagem de consulta em instruções de baixo nível, inteligíveis ao componente de execução de consultas. Além disso, o compilador DML tenta transformar a solicitação do usuário em uma solicitação equivalente, mas mais eficiente, buscando, assim, uma boa estratégia para a execução da consulta.
- **Pré-compilador para comandos DML:** inseridos em programas de aplicação, que convertem comandos DML em chamadas de procedimentos normais da linguagem

hospedeira. O pré-compilador precisa interagir com o compilador DML de modo a gerar o código apropriado.

- **Interpretador DDL**: interpreta os comandos DDL e registra-os em um conjunto de tabelas que contêm metadados.
- **Componentes para o tratamento de consultas**: executam instruções de baixo nível geradas pelo compilador DML.

Os componentes para administração do armazenamento de dados proporcionam a interface entre os dados de baixo nível, armazenados no banco de dados, os programas de aplicações e as consultas submetidas ao sistema. Os componentes de administração de armazenamento de dados incluem:

- **Gerenciamento de autorizações e integridade**: testam o cumprimento das regras de integridade e a permissão ao usuário no acesso ao dado.
- **Gerenciamento de transações**: garantem que o banco de dados permanecerá em estado consistente a despeito de falhas no sistema e que transações concorrentes serão executadas sem conflitos em seus procedimentos.
- **Administração de arquivos**: gerencia a alocação de espaço no armazenamento em disco e as estruturas de dados usadas para representar estas informações armazenadas em disco.
- **Administração de buffer**: responsável pela intermediação de dados do disco para a memória principal e pela decisão de quais dados colocar em memória cache.

Além disso, algumas estruturas de dados são exigidas como parte da implementação física do sistema:

- **Arquivo de dados**: armazena o próprio banco de dados.

- **Dicionário de dados**: armazena os metadados relativos à estrutura do banco de dados. O dicionário de dados é muito usado. Portanto, grande ênfase é dada ao desenvolvimento de um bom projeto com uma implementação eficiente do dicionário.
- **Índices**: proporcionam acesso rápido aos itens de dados que são associados a valores determinados.
- **Estatísticas de dados**: armazenam informações estatísticas relativas aos dados contidos no banco de dados. Essas informações são usadas pelo processador de consultas para seleção de meios eficientes para execução de uma consulta.

A Figura 5.2 mostra esses componentes e a conexão entre eles.

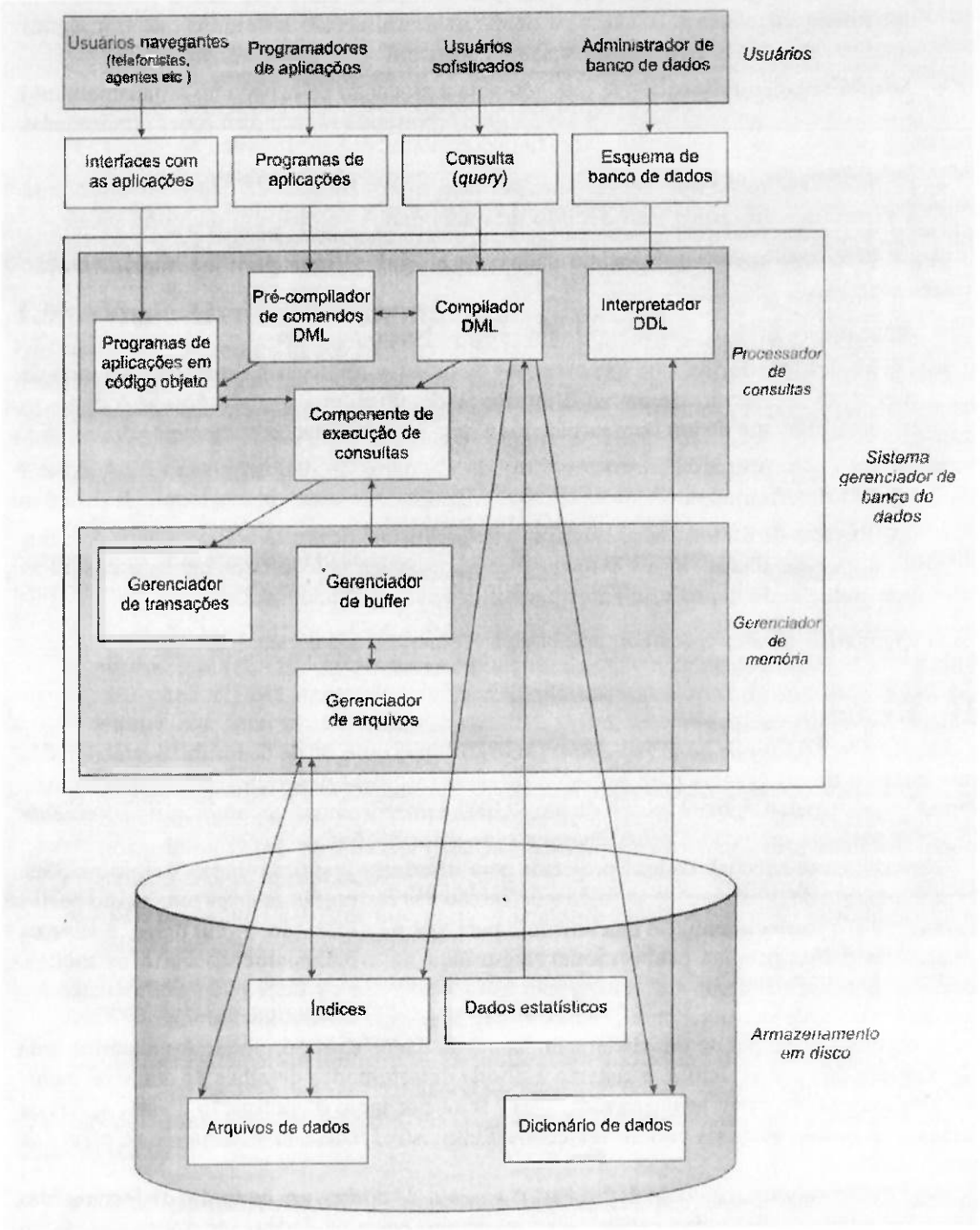


Figura 5-2 – Sistema de banco de dados [SILBERSCHATZ, 1997], [KORTH, 1997], [SUDARSAHN, 1997]

O objetivo principal de um sistema de banco de dados é proporcionar aos usuários uma visão abstrata dos dados. Isto é, o sistema esconde determinados detalhes de como os dados são mantidos e como estão

armazenados. Isto é feito por meio da definição de três níveis de abstração de dados, os quais podem ser vistos como: nível físico, nível lógico e os níveis de visão.

Sob a estrutura do banco de dados está o modelo de dados: um conjunto de ferramentas conceituais para descrição dos dados, relacionamento entre os dados, semântica dos dados e suas restrições. Os vários modelos de dados propostos possuem três diferentes grupos: modelos lógicos orientados a objetos, modelos lógicos com base em registros e modelos físicos de dados.

Os bancos de dados mudam ao longo do tempo pela inserção e exclusão de informações. O conjunto de informações armazenadas no banco de dados, em determinado momento, é chamado instância do banco de dados. O projeto do banco de dados como um todo é chamado esquema. A capacidade de modificar a definição de um esquema em um nível, sem afetar a definição do esquema do nível superior, é chamada independência de dados. Existem dois níveis de independência de dados: físico e lógico. Um esquema de banco de dados é determinado por um conjunto de definições que são expressas pela linguagem de definição de dados (DDL). Os comandos DDL são compilados em um conjunto de tabelas armazenadas em um arquivo especial chamado dicionário de dados; portanto, o dicionário de dados possui metadados.

Uma linguagem de manipulação de dados (DML) é a linguagem que habilita o acesso e manipulação dos dados ao usuário. Existem basicamente dois tipos: DMLs procedurais, que obrigam a especificação de quais dados são necessários e como obtê-los; e DMLs não procedurais, que exigem que o usuário especifique quais dados serão necessários, sem especificar como estes dados serão obtidos.

O gerenciador de informações é responsável por garantir a manutenção de estados consistentes (correção), a despeito de falhas no sistema. O gerenciador de transações também garante que transações concorrentes sejam executadas sem conflito. O gerenciador de memória é um módulo de programa que proporciona a interface entre o armazenamento dos dados em baixo nível e os programas de aplicação e consulta. O gerenciador de memória é responsável pela interação com os dados armazenados em disco.

5.3. Sistema Cliente-Servidor

O conceito de Cliente-Servidor está no entendimento da relação lógica entre uma entidade que requisita o serviço (Cliente) de uma outra entidade que disponibiliza um serviço requisitado (Servidor).

O cliente e o servidor podem ou não existir em máquinas distintas. Um Cliente pode ter uma relação com diferentes servidores e um servidor pode satisfazer requisições de múltiplos clientes. A relação entre um cliente e um servidor é conduzida por "transações" constituindo uma relação bem definida de requisições e respostas. Uma Vantagem do sistema Cliente-Servidor é a divisão do processamento entre eles, como resultado, a relação entre eles é conhecida como Processamento Cooperativo [RENAULD, 1996].

O processo Cliente-Servidor basicamente consiste em um Cliente solicitar uma requisição para um Servidor e este responder com resultados para as requisições. Como o próprio nome do sistema indica, o Servidor processa os serviços provenientes de seus Clientes, usualmente por um caminho de processamento especifico que somente os clientes têm acesso.

Fisicamente, tanto o cliente, quanto o servidor são indiferentes a estarem na mesma ou em diferentes máquinas. Alguns IPC (Comunicação entre processos) ou protocolos de redes podem não serem suportados pelo cliente ou pelo servidor no mesmo sistema. Entretanto, a restrição de um protocolo em particular na caracteriza o Sistema Cliente-Servidor. Do ponto de vista da máquina, normalmente, tem-se a impressão que o servidor é uma máquina de maior potência, porém não há nenhuma razão de o servidor não ser executado num computador de pequeno porte enquanto o cliente estar situado num computador de grande porte. A Figura 5.3 ilustra os vários tipos de sistemas Cliente-Servidor.

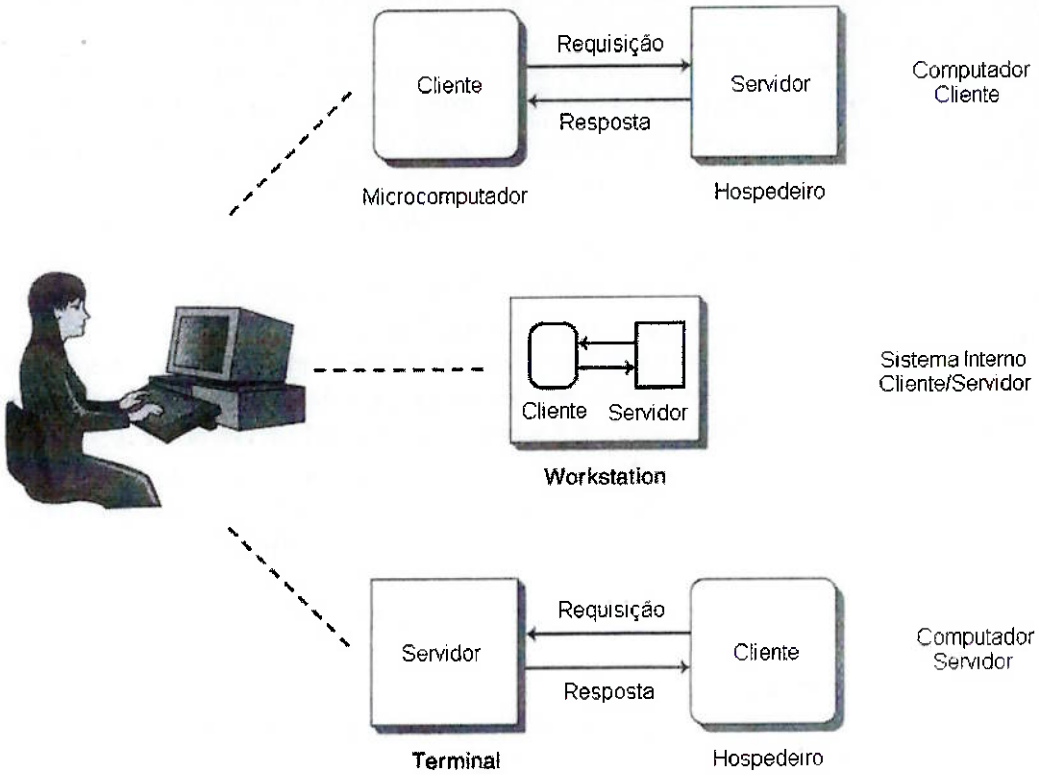


Figura 5-3 – Sistema cliente-servidor [RENAULD, 1996]

A simplicidade do conceito do sistema Cliente-Servidor torna esse tipo de sistema um dos mais utilizados atualmente. Esse sistema é a base para os sistemas voltados a Web (Internet e Intranet).

6. APLICAÇÕES DE TÉCNICAS DE TESTES DE SOFTWARE

Este capitulo tem por objetivo relatar as aplicações de técnicas de testes softwares as principais arquiteturas de sistema.

6.1. Sistemas Tempo-Real

Neste tipo de sistema, encontram-se o elemento de combinação alinhado ao teste que é o tempo. Os testes de software devem levar em consideração o impacto das falhas de hardware sobre o processamento do software. Tais falhas podem ser extremamente difíceis de ser simuladas realísticamente.

Para os sistemas de Tempo-Real é utilizada uma estratégia de testes:

- **Teste de Tarefas**: O primeiro passo da atividade de testes de tempo-real consiste em testar cada tarefa independentemente. Este teste revela erros de lógica e de função, mas não revelará erros comportamentais ou de relacionados com o tempo.
- **Teste comportamental**: É possível simular o comportamento de um sistema de tempo-real e examinar seu comportamento como uma conseqüência de eventos externos. O comportamento do software é examinado a fim de detectar erros de comportamento.

6.1.1. Teste de Tarefas

Devido à complexidade e todos os possíveis relacionamentos entre variáveis, algumas técnicas de software necessitam ser combinadas a fim de se obter o resultado desejado.

Park e Shaw (1991) desenvolveram um programa capaz de analisar outros programas escritos em uma versão restrita da linguagem C, que inclui operadores aritméticos e relacionais simples (que não fazem chamadas a bibliotecas), atribuições, chamadas de funções, *arrays*, ponteiros, estruturas e as construções *if* e *while*. A entrada para o programa é o código fonte em C. O número máximo e mínimo de iterações de cada laço é verificado (Teste de Laço), assim como o fluxo de dados é observado (Teste de Fluxo de Dados) no sistema.

A ferramenta produz resultados bastante precisos para programas simples, porém, programas complexos podem possuir caminhos irrealizáveis (uma seqüência de segmentos que nunca é executada). Quando estes caminhos são considerados, podem produzir resultados muito abaixo ou acima dos reais. Por exemplo: em uma rotina de escalonamento é dividida em três sub-rotinas: procura por processos em condição de execução, alocação do um processador para estes processos e preempção [PARK, 1991].

6.1.2. Teste Comportamental

Um programa que possui exigências temporais na sua especificação só pode ser considerado correto se não apresentar falhas na sua lógica nem no prazo da conclusão das tarefas. Em engenharia de software, há uma série de métodos que visam assegurar a correção lógica do programa. Por

outro lado, a correção temporal exige técnicas especiais, muitas ainda abertas para pesquisa [BRAGA, 1999].

Técnicas de projeto direcionadas para a área de Sistemas de Tempo-Real procuram incluir as restrições de tempo logo nas primeiras etapas de desenvolvimento. Entretanto, muitas linguagens de programação não oferecem recursos específicos para assegurar que estas restrições estão sendo cumpridas durante a fase de implementação. Por esta razão, é preciso comprovar que o resultado final cumpre as exigências temporais.

O tempo de execução das instruções da maioria dos processadores é fornecido pelos Fabricantes. A partir destes valores é possível calcular o tempo de execução de um programa somando os tempos das instruções individuais. A presença de desvios condicionais torna o cálculo complexo pois os diferentes caminhos que a execução pode tomar devem ser considerados, gerando equações que relacionam o tempo a valores de variáveis.

Todo programa executável pode ser representado por um grafo de controle. Um segmento consiste em um trecho de programa executável (neste contexto os programas executáveis serão sempre considerados como um conjunto de instruções do processador) cuja característica determinante é a sua execução seqüencial, ou seja, sempre que o processador iniciar a execução de um segmento o mesmo será executado até o fim. Portanto, um segmento não pode conter instruções de salto, exceto a última.

O grafo de controle é um grafo direcionado cujos nós representam os segmentos do programa. Os possíveis caminhos a seguir após o término da execução de um segmento são representados pelos arcos, melhor representados na figura 6.1. Como os segmentos têm execução seqüencial, é possível calcular um valor numérico para seu tempo.

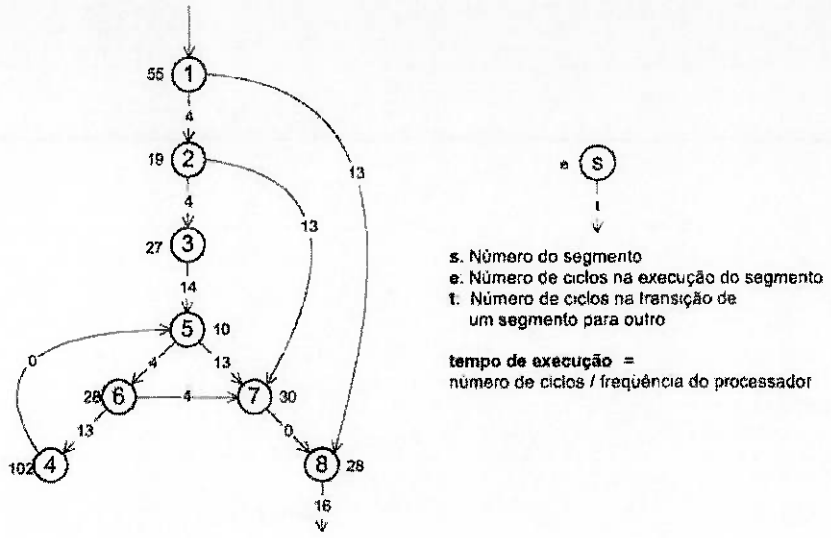


Figura 6-1 – Grafo de controle [BRAGA, 1999]

Considerando que qualquer programa não trivial tem um número elevado de segmentos, estes podem ser combinados em macro-segmentos, facilitando o entendimento e a visualização. Um macro-segmento consiste de um conjunto de segmentos interligados. Um macro-segmento possui um único ponto de entrada e todas as suas saídas (arcos para nós fora do macro-segmento) são direcionadas para um mesmo nó. A definição de macro-segmento é recursiva, ou seja, um conjunto de macro-segmentos com um único ponto de entrada e todas as saídas direcionadas para um mesmo nó também é um macro-segmento(ver Figura 6.2). Assim pode-se representar um programa como uma árvore cuja raiz é um macro-segmento (que representa o programa todo).

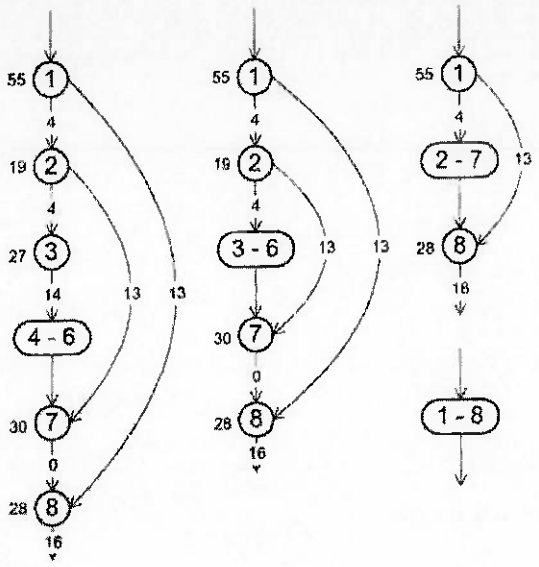


Figura 6-2 – grafo de macro-segmentos [BRAGA, 1999]

O grafo de controle também pode apresentar os tempos de execução medidos durante a execução de um programa. Neste caso, os valores são obtidos através de uma ferramenta de medição enquanto o sistema executa vetares de teste. Programas escritos em linguagens de alto nível contém um grande número de segmentos introduzidos pelo compilador. Neste caso, a análise do grato de segmentos do torna-se muito complexa e pode-se utilizar o grato gerado diretamente a partir do código em alto nível.

6.2. Sistemas Centrados em Banco de Dados

Para abordagem de Aplicações de Técnicas de Teste de Software para Sistemas Centrado em Banco de Dados, os métodos de Testes Estruturais foram estendidos e adaptados. Estas adaptações permitem uma melhor interpretação dos Grafos de Fluxo de Controle e Grafos de Fluxo de Dados. Para sua posterior aplicação em suas próprias técnicas e nas técnicas de Caminho Básico, Particionamento de Equivalência e Análise de Valor Limite.

6.2.1. Critérios baseados no Fluxo de Controle

Na abordagem de testes para Aplicações de Banco de Dados Relacional, estende-se a representação do Grafo de Fluxo de Controle, em que os nós arredondados representarão blocos de comandos da linguagem de programação e os nós retangulares representarão comandos SQL [SPOTO, 2000].

Portanto, a notação utilizada para representar o Grafo de Fluxo de Controle de Aplicações de Banco de Dados Relacional é representada na figura 6.3.

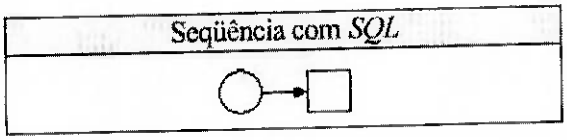
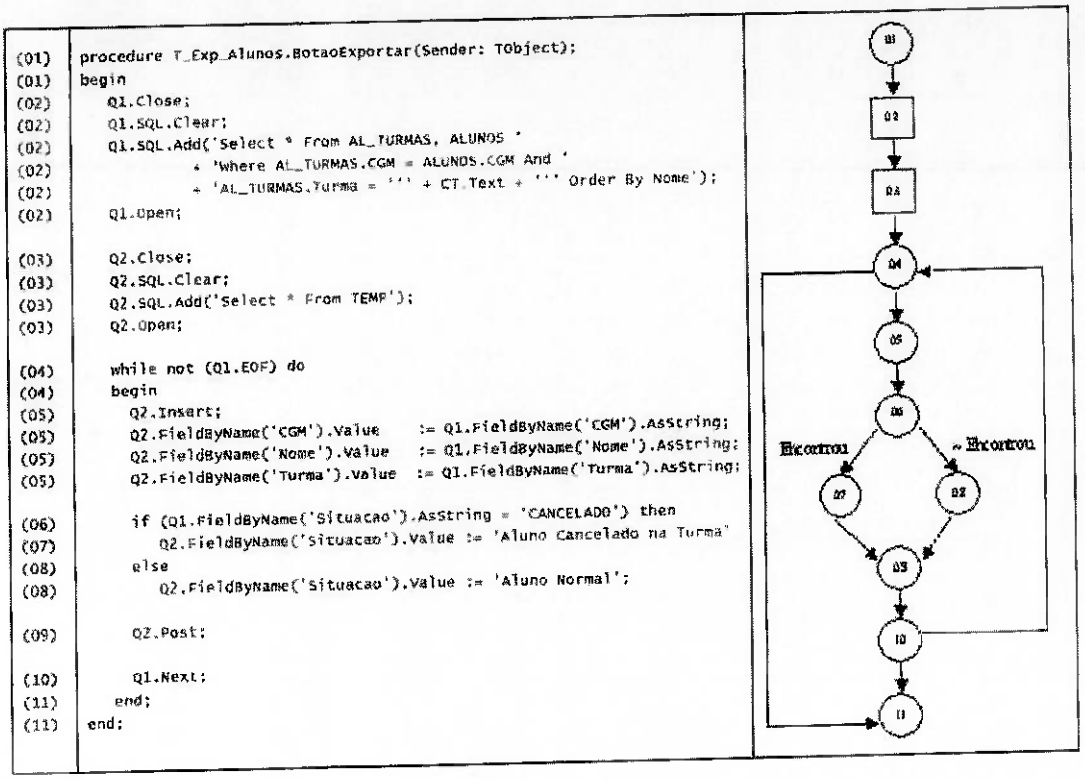


Figura 6-3 – Notação de grafo de fluxo de controle para aplicação de banco de dados relacional [GONÇALVES, 2003]

O exemplo mostrado na figura 6.4 apresenta os critérios de cobertura de Fluxo de Controle de Aplicação de Banco de Dados Relacional que tem por objetivo de exportar dados dos alunos de uma faculdade [GONÇALVES, 2003]



```
(01)  procedure T_Exp_Alunos.BotaoExportar(Sender: TObject);
(01)  begin
(02)     Q1.Close;
(02)     Q1.SQL.Clear;
(02)     Q1.SQL.Add('Select * From AL_TURMAS, ALUNOS '
(02)              + 'Where AL_TURMAS.CGM = ALUNOS.CGM And '
(02)              + 'AL_TURMAS.Turma = ''' + CT.Text + ''' Order By Nome');
(02)     Q1.Open;

(03)     Q2.Close;
(03)     Q2.SQL.Clear;
(03)     Q2.SQL.Add('Select * From TEMP');
(03)     Q2.Open;

(04)     while not (Q1.EOF) do
(04)     begin
(05)       Q2.Insert;
(05)       Q2.FieldByName('CGM').Value     := Q1.FieldByName('CGM').AsString;
(05)       Q2.FieldByName('Nome').Value    := Q1.FieldByName('Nome').AsString;
(05)       Q2.FieldByName('Turma').Value   := Q1.FieldByName('Turma').AsString;

(06)       if (Q1.FieldByName('Situacao').AsString = 'CANCELADO') then
(07)          Q2.FieldByName('Situacao').Value := 'Aluno Cancelado na Turma'
(08)       else
(08)          Q2.FieldByName('Situacao').Value := 'Aluno Normal';

(09)       Q2.Post;

(10)       Q1.Next;
(11)     end;
(11)  end;
```


Figura 6-4 – Grafo de fluxo de controle de aplicação de banco de dados relacional [GONÇALVES, 2003]

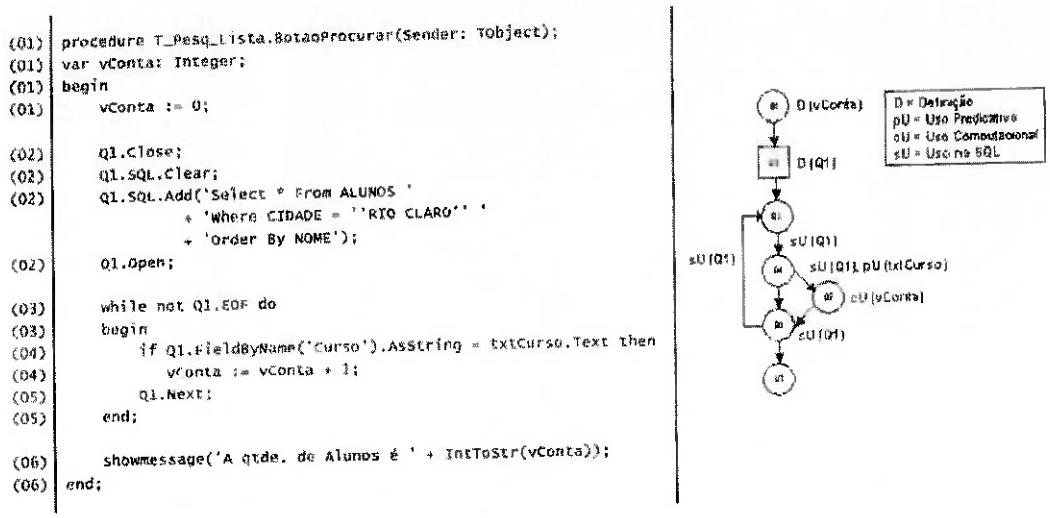
6.2.2. Critérios baseados no Fluxo de Dados

Segundo Spoto (2000), em uma aplicação de Banco de Dados, as variáveis utilizadas podem ser classificadas em três tipos:

- Variáveis de Programa: variáveis definidas e usadas apenas na Linguagem Hospedeira;
- Variáveis Hosts: também conhecidas como Variáveis de Ligação, são definidas e usadas em qualquer parte do Módulo de Programa (dentro e fora da SQL) e são fundamentais para estabelecer a comunicação entre as duas linguagens;
- Variáveis de Tabela: definidas e usadas apenas nos comandos executáveis da SQL.

Os comandos DML (SELECT, INSERT, UPDATE e DELETE) caracterizam a ocorrência de definição da Variável de Tabela.

Além das definições de uso de uma variável c-uso e p-uso, para a abordagem de Aplicações Banco de Dados Relacional, a referência s-uso (uso na SQL): quando ao variável afetar o resultado de uma consulta ou de uma computação em um comando executável da SQL (Uso associado ao nó da SQL).



```
(01) procedure T_Pesq_Lista.BotaoProcurar(Sender: TObject);
(01) var vConta: Integer;
(01) begin
(01)     vConta := 0;

(02)     Q1.Close;
(02)     Q1.SQL.Clear;
(02)     Q1.SQL.Add('Select * From ALUNOS '
                  + 'Where CIDADE = ''RIO CLARO'' '
                  + 'Order By NOME');
(02)     Q1.Open;

(03)     while not Q1.EOF do
(03)     begin
(04)         if Q1.FieldByName('Curso').AsString = txtCurso.Text then
(04)            vConta := vConta + 1;
(05)         Q1.Next;
(05)     end;

(06)     showmessage('A qtde. de Alunos é ' + IntToStr(vConta));
(06) end;
```


Figura 6-5 – Grafo de fluxo de dados de uma aplicação de banco de dados relacional [GONÇALVES, 2003]

6.2.3. Testes Estruturais e Testes Funcionais

Para a aplicação das técnicas de Caixa Branca podem ser utilizados critérios baseados em Fluxo de Controle / Complexidade Ciclomática, a partir da aplicação do Teste de Caminhos Básicos, que permite definir uma maneira de determinar um conjunto básico de caminhos linearmente independentes, de modo que os executando, garante-se a execução de todos os comandos ao menos uma vez. Já os critérios baseados em Fluxo de Dados estabelecem requisitos de teste que seguem o modelo de dados

usado dentro do programa. Deve-se, ainda, levar em consideração, a extensão proposta por Spoto (2003), para Aplicações de Banco de Dados Relacional.

No entanto, a escolha por técnicas de Teste Funcional permite por à prova a especificação do sistema. Para isso, podem ser utilizadas as técnicas de Particionamento de Equivalência e Análise de Valor Limite. A primeira técnica citada possibilita a redução do tamanho do domínio de entrada, pois se concentra em criar dados de teste baseados unicamente na especificação, e é indicado principalmente para especificação de aplicações, onde as variáveis de entrada podem ser detectadas mais facilmente. Complementando esta técnica, a Análise de Valor Limite coloca sua atenção em uma fonte propícia a erros, os limites de uma classe ou partição de equivalência. Portanto, estas técnicas se aplicadas em conjunto permitem um melhor aproveitamento para a análise dos Casos de Testes que devem ser gerados. Além disso, a estratégia proposta por Binder (2000), permite uma adaptação para a especificação do Modelo Entidade-Relacionamento, possibilitando assim, testes nas especificações de Aplicações de Banco de Dados Relacional.

6.3. Sistemas de Cliente-Servidor

No ambiente cliente-servidor, uma grande dificuldade do teste é a divisão em camadas, o que aumenta a complexidade do software e dificulta a atividade de teste. Por isso são utilizados os testes de unidade, utilizando técnicas e critérios convencionais, executam-se testes para componentes, métodos e funções que sejam mais complexas e cujo grau de prioridade seja alto. Testes de integração devem ser feitos conforme a necessidade e complexidade do sistema e os componentes devem ser testados a partir do término do desenvolvimento dos primeiros casos de uso com teste de

estresse, performance, integridade e segurança. Esses testes são feitos com as três camadas e, quando os resultados são duvidosos, deve-se isolar as camadas. [VOLPI, 2002], [VERGÍLIO, 2002].

Segundo Mosley (2000), os testes em ambiente cliente-servidor devem levar em consideração cada camada isoladamente. Exemplos de teste para a arquitetura três-camadas são:

- **Camada de apresentação**: devem ser realizados teste de usabilidade, teste de intuitividade de ícones, etc.
- **Camada do servidor**: devem ser realizados teste de carga e volume, teste de estresse, teste de desempenho, teste de recuperação ou devolução de dados, teste de segurança dos dados, teste de integridade dos dados, etc.
- **Camada de aplicação**: nesta camada, os componentes executam toda a lógica de negócio da aplicação. Devem ser realizados muitos dos testes apontados na camada do servidor.

6.3.1. Camada de Apresentação

A camada de apresentação, como o próprio no diz, é responsável pela visualização dos dados obtidos no servidor ao cliente. Nesta camada , portanto, temos basicamente o teste da GUI (*Graphic User Interface* – Interface Gráfica com o Usuário).

Para o teste da GUI são utilizadas as mais tradicionais técnicas de teste de software. Alguns dos mais sérios erros nos sistemas de software tem sido os resultados de inadequadas ou uma falta total de processos de validação. Os testes de software que tem se mostrado mais adequado a

essas validações são os testes de caixa-preta, mas especificadamente as técnicas de Particionamento de Equivalência e Análise de Valor Limite.

Em complemento a estas técnicas tradicionais de teste de software, muitas organizações estão desenvolvendo ferramentas de "captura" para testar a usabilidade das GUI. A diferença entre as técnicas tradicionais e as novas ferramentas desenvolvidas é que a "captura" ocorre em um nível interno do sistema. As entradas de dados, seja pelo *mouse* ou pela digitação, e as saídas na tela do cliente são registradas e comparadas com os resultados previamente esperados, possibilitando assim, a verificação da usabilidade da interface.

6.3.2. Camada de Servidor

Na camada de servidor temos a realização de muitos testes específicos, no qual, não é possível relacionar técnicas comuns de software como nas diferentes camadas. Como não se trata do escopo do trabalho, os testes realizados nesta camada serão apenas citados os principais teste realizados quanto a sua funcionalidade.

6.3.2.1. Teste de Carga

Os sistemas Cliente-Servidor devem ser submetidos a dois tipos de testes: o teste funcional baseado em um único usuário e o teste de carga com múltiplos usuários.

O teste de carga com múltiplos usuários é o melhor método para medir a performance do sistema Cliente-Servidor. É necessário para determinar a melhor adequação da aplicação do servidor, do banco de

dados do servidor e o desempenho de aplicações *Web*. Como o teste de múltiplos usuários requer uma situação na qual múltiplos clientes acessam uma única aplicação no servidor é impossível de se realizar sem uma automação prévia do sistema.

Para o teste de carga alguns objetivos devem ser observados:

- Medir o tempo completo para realização da tarefa requisitada;
- Descobrir qual configuração de software e hardware otimiza a performance;
- Verificar as consultas do Banco de dados que otimizam o sistema;
- Capturar o tempo de resposta de erro.

6.3.2.2. Teste de Volume

O objetivo do teste de volume é descobrir o limite do sistema no que diz respeito à exposição de um grande número de dados durante a um extenso período de tempo.

6.3.2.3. Teste de Estresse

O objetivo do teste de estresse é achar falhas da capacidade do sistema no que diz respeito à exposição de um grande número de transações em um curto espaço de tempo.

6.3.2.4. Teste de Desempenho

O teste de desempenho é geralmente realizado quanto ao tempo de resposta e acessos de transações sob diferentes processamentos e condições de configuração.

Segundo Hamliton (1994), os problemas de desempenho são mais freqüentes devido à configuração do cliente ou do servidor.

A melhor estratégia para realizar o teste de performance é baseada em três passos:

- Executar um teste controlado de dados, observando volume, estresse e carga;
- Analisar esses dados;
- Examinar as consultas de acesso ao banco de dados e, se necessários, providenciar um armazenamento temporário no cliente enquanto as aplicações no servidor estiverem sendo executada.

6.3.3. Camada de Aplicação

A Camada de Aplicação é onde está a inteligência do sistema. Devido a isso, os programas são testados de diferentemente das demais camadas do sistema.

Na Camada de Aplicação não se tem a preocupação com a interface com o usuário como na camada de apresentação ou com as estruturas do sistema como na camada de servidor.

Nesta Camada, as principais funções a nível funcional são a verificação dos acessos das transações pela requisição do cliente e a resposta do servidor. A nível estrutural, como as transações são, geralmente, centradas em banco de dados, o teste mais utilizado é o mesmo dos sistemas centrados em banco de dados – Teste de Fluxo de dados.

6.4. Resumo

A Tabela 6.1 é um resumo das técnicas de teste relacionadas com as arquiteturas de sistemas.

Sistema	**Teste**
Tempo-Real	Teste de Laço
	Teste de Fluxo de Dados
	Teste de Fluxo de Controle
Centrado em Banco de Dados	Particionamento de Equivalência
	Análise de Valor Limite.
	Teste de Fluxo de Dados
	Teste de Fluxo de Controle
Cliente/Servidor	Particionamento de Equivalência
	Análise do Valor Limite
	Teste de Fluxo de Dados

Tabela 6-1 – Resumo da Técnicas de Teste X Sistema

7. CONCLUSÃO

O teste é muito importante para a garantia da qualidade dos sistemas produzidos ou gerenciados por qualquer empresa. O teste é necessário, para que se atinjam os níveis mais altos de qualidade e o grau de certificação que toda empresa almeja e para que seus produtos sejam formalmente testados com critério. Um teste feito ao acaso estará fadado ao insucesso quase inevitável. O planejamento não é a garantia de que tudo correrá tranqüilamente, mas é um passo necessário que aumenta muito as chances de sucesso.

Uma estratégia de teste de software integra técnicas de projeto de casos de teste numa série bem definida de passos que resultam na construção bem sucedida do software, Esses passos podem envolver uma combinação das técnicas e aplicação de diferentes critérios de teste. Geralmente, a escolha de um critério de teste é guiada pelos fatores eficácia (número de defeitos revelados), e custo (número de casos de teste requeridos). No entanto, as técnicas e critérios devem ser aplicados em diferentes etapas do teste e são considerados complementares, pois podem revelar diferentes classes de defeitos. Essas técnicas dizem respeito à etapa de projeto de casos de teste, entretanto, é importante executar todos os passos da atividade de teste e não reduzir esse prazo em detrimento de outros problemas.

Portanto, não se pode afirmar que uma determinada técnica de teste de software obtenha um melhor resultado ou tenha uma maior eficiência ou ainda, que seja mais completa em relação às demais técnicas apresentadas neste trabalho.

Pode se afirmar que para cada diferente arquitetura de sistema, há uma série de técnicas de teste de software que melhor se adaptam e resultam em melhores resultados do que outras. Esses resultados foram

mostrados por relatos publicados. Porém, dentro de uma mesma arquitetura pode-se haver técnicas distintas dependendo da fase do desenvolvimento do sistema que o software é submetido ao teste.

Na verdade, a melhor técnica de teste é aquela que melhor se adapta ao sistema em determinada fase do teste.

Como sugestão de continuação do estudo realizado seguem duas possíveis propostas:

- Estudo de outras técnicas de teste de software e de outras arquiteturas de sistema e suas respectivas aplicações;
- Estudo individualizado das aplicações das técnicas de teste de software em cada diferente arquitetura de sistema apresentadas neste trabalho, aprofundando-se nos teste das distintas fases de teste no desenvolvimento do sistema (Unidade, Integração, Validação e Sistema).

O trabalho realizado foi de extrema importância para o autor tanto do ponto de vista profissional como acadêmico. Do ponto de vista profissional, pode adquirir um maior conhecimento sobre a área de teste de software e notar como a disciplina é abrangente. Do ponto de vista acadêmico, pode realizar uma atividade de pesquisa sobre um tema específico, analisar diferentes fontes e estruturar um trabalho.

8. RERERÊNCIAS BIBLIOGRÁFICAS

- Adrion, W. R.; Bransted, M. A.; Cherniavsky, J. C. Validation, Verification ans Testing of Computer Software **ACM Computing Surveys**, v.14, n.2, p.159-192, jun. 1982

- Beizer, B. **Software Testing Techniques**. New York: Van Nostrand Reinhold, 1983

- Beizer, B. **Software System Testing and Quality Assurance**. New York: Van Nostrand Reinhold, 1984.

- Beizer, B. **Black-Box Testing: Techniques for Functional Testing of Sofware and Systems**. John Wiley & Sons, 1995.

- Bochenski, B. **Implementando sistemas cliente/servidor de qualidade**. Makron Books, 1995.

- Braga, A. S.; Renaux, D. P. B. **Utilização de Linhas de Tempo para Depuração e Validação Temporal de Sistemas em Tempo Real**. I Workshop em Sistemas de Tempo-Real, 1998.

- Braga, A. S. **TLM – Uma ferramenta de apoio ao teste de restrições temporais em sistemas dedicados operando em Tempo-Real**, Curitiba, 1999.

- Conway, B. **Testing Client/Server Applications: challenges, strategies and solutions for success**, Stamford: Gartner Group, 1996.

- Delta-4, **Real-Time Concepts**, on Delta-4 Architecture Guide, Cap. 5, 1991

- Gelperin, D.; Hetzel B. The Growth of Software Testing. **Communications of the ACM**, v.31, n.6, p. 687-695, jun. 1988.

- Gonçalves, K. V. **Teste de Sofware em Aplicações de Banco de Dados Relacional**, Campinas, 2003.

- Hetzel, B. **The complete guide to software testing**. QED Information Sciences, 1984.

- Hamilton, D. **Don't Let Client/Server Performance Gotchas Getcha**. Datamation V. 40, n. 21, 1994.

- Howden, W. E. A Survey of Dynamic Analysis Methods. **Tutorial: Software Testing & Validation Techniques**, IEEE Computer Society Press, catálogo n. EHO 180-0, p 209-231, 1981

- Miller, E.; Howden, W. E. **Tutorial: Software Testing & Validation Techniques**. IEEE Computer Society, 1981.

- Mosley, D. J. **Client/Server Software Testing on the Desktop and the Web**. Prenctice Hall, 2000.

- Myers, G. J. **The Art of Software Testing**. John Wiley & Sons, 1979.

- Park, C. Y.; Shaw A. C. Experiments with a Program Timing Tool Based on Source-Level Timing Schema **Computer – IEEE Computer Society**. V. 24, n. 5 - 1991

- Perry, W. **Effective Methods for Software Testing**. John Wiley & Sons, 1995.

- Pressman, R. S. **Software Engineering: A Practitioner's Approach**. McGraw-Hill, 1992.

- Rota, S. R. **Um método de teste de software baseado em casos de testes.** 2001. 173p. Dissertação (Mestrado) – Escola Politécnica. Universidade de São Paulo. São Paulo, 2001.

- Renauld, P. E. **Introduction to Client/Server Systems** Canada, 1996

- Silberschatz, A.; Korth, H. F.; Sudarshan S. **Sistema de Banco de Dados**, 1997

- Spoto, E.; Jino, M.; Maldonado, J. C. **Teste Estrutural de Software: Uma abordagem para Aplicações de Banco de Dados Relacional**, João Pessoa: XIV Simpósio Brasileiro de Engenharia de Software, 2000.

- Stankovic, J. A.; Ramamrithan K. What is predictability for Real-Time Systems ? – **The Journal of Real Time Systems,** vol.2, p. 247-254, 1990

- Tai, K. C. Predicate-Based Test Generation for Computer Programs. **Proceedings of the 15th Internacional Conference on Software Egineering**, p267-276, 1993

- Volpi, L. M.; Vergilio, S. R. **Teste em ambiente Cliente-Servidor: Uma estratégia e um estudo de caso**, Bate Byte, 2002

- Weyuker, E. J. Testing Component – Based Software: A Cautionary Tale. **IEEE Software**, v.15, n.5, p.54-59, 1998

- White, L. J.; Cohen, E. I. A Domain Strategy for Computer Program Testing. **IEEE Transactions on Software Engineering**, v.6, n.3, p.247-257, 1980