

Paulo Augusto Nardi

Inclusão do Critério Todos-T-Usos na Ferramenta JaBUTi

MARÍLIA/SP

2006

Paulo Augusto Nardi

Inclusão do Critério Todos-T-Usos na Ferramenta JaBUTi

Banca examinadora da dissertação apresentada ao Programa de Mestrado da UNIVEM, /F.E.E.S.R., para obtenção do Título de Mestre em Ciência da Computação.

RESULTADO:

ORIENTADOR: Márcio Eduardo Delamaro

CO-ORIENTADOR: Edmundo Sérgio Spoto

1. EXAMINADOR:

2. EXAMINADOR:

3. EXAMINADOR:

Marília, ___ de _____ de 2006

Agradeço aos meus orientadores Márcio Eduardo Delamaro e Edmundo Sérgio Spoto pelos ensinamentos, orientações, paciência e liberdade de escolha.

À minha esposa, Joice de Siqueira Alves, pelo apoio incondicional, pela compreensão e por ensinar que o amor não está limitado, e muito menos é abalável, pela distância que nos separou por estes anos.

Aos meus amigos Rodolfo Barros Chiaramonte e Larissa Pavarini, pelo companheirismo, sugestões e momentos de descontração que permitiram que esta jornada se tornasse mais agradável.

Aos demais professores, pelo conhecimento passado.

À Capes, pelo suporte financeiro.

*"...The important thing is this: to be able at any moment to sacrifice what we are for
what we could become."*

"Da Escola de Guerra da Vida - o que não me mata torna-me mais forte."

Resumo

Esta dissertação apresenta os principais conceitos que envolvem a inclusão de um critério de aplicação de banco de dados relacional (ABDR), todos-t-usos, na Ferramenta JaBUTi (Java Bytecode Understanding and Testing). Os critérios ABDR pertencem à técnica estrutural de teste e permitem a realização de testes de unidade e integração com base no fluxo de dados, considerando-se a relação entre as tabelas acessadas na aplicação. A Ferramenta JaBUTi permite a automatização de determinados processos da fase de teste estrutural para programas em Java e fornece meios para uma melhor compreensão do código e a análise sobre os testes realizados. É apresentada uma solução de implementação das características relacionadas ao critério todos-t-usos na JaBUTi. Como avaliação da Ferramenta apresentada, é realizado um estudo de caso e apresentados os seus resultados e análises.

Abstract

This thesis presents the main concepts related to the inclusion of a relational database application criterion (RDA), all-t-uses, in the JaBUTi Tool. RDA criteria belongs to the structural technique of test and allow the accomplishment of unit and integration tests based on the data flow, considering the relation between tables accessed by the application. The JaBUTi tool allows the automatization of many processes of structural test for programs in Java and supplies ways to a better understanding of the code and the analysis on the accomplished tests. A solution of implementation of the characteristics related to the criterion all-t-uses in the JaBUTi is presented. As evaluation of the presented Tool, a case study is performed and presented its results and analyses.

Sumário

	p.17
1.1 Motivação	p.18
1.2 Objetivo	p.19
1.3 Organização	p.19
	p.20
2.1 Conceitos Básicos	p.20
2.2 Técnica de Teste Estrutural	p.23
2.2.1 Grafos de Programa	p.23
2.2.2 Critérios de teste baseados em fluxo de controle	p.25
2.2.3 Critérios de teste baseados em fluxo de dados	p.30
2.3 Análise de Inclusão	p.33
2.4 Conclusão	p.35
	p.36
3.1 Conceitos Básicos	p.36

3.1.1	Distribuição de Peso para o Critério Todos-Nós	p. 38
3.1.2	Distribuição de Peso para o Critério Todos-Arcos	p. 41
3.1.3	Distribuição de Peso para o Critério Todos-Usos	p. 42
3.2	Compreensão do Código	p. 42
3.2.1	Grafo	p. 42
3.2.2	Visualização dos Elementos Requeridos	p. 45
3.2.3	Análise do Código	p. 46
3.3	Realização de Teste com a Ferramenta JaBUTi	p. 47
3.4	Conclusão	p. 54
4	Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional	p. 55
4.1	Conceitos Básicos	p. 55
4.2	Crterios de Teste	p. 57
4.3	Casos de Teste	p. 58
4.4	Exemplo Prático	p. 58
4.4.1	Análise da Aplicação e Base de Dados	p. 59
4.4.2	Criação e Análise dos Casos de Teste	p. 61
4.5	Análise de Inclusão	p. 64
4.6	Conclusão	p. 64
5	Análise de ABDR pela JaBUTi	p. 70
5.1	Visão Geral das Características Implementadas	p. 70
5.1.1	Análise Estática	p. 70

5.1.2	Análise Dinâmica	p. 71
5.2	Características e Aspectos de Implementação na Fase Estática	p. 72
5.2.1	Elementos Requeridos do Critério All-T-Uses-ei	p. 78
5.2.2	Elementos Requeridos do Critério All-T-Uses-ed	p. 80
5.2.3	Distribuição de Pesos	p. 81
5.3	Características e Aspectos de Implementação na Fase Dinâmica	p. 85
5.4	Conclusão	p. 87
6	Estudo de Caso	p. 89
6.1	Descrição do Sistema	p. 89
6.1.1	Aplicação do Critério Todos-T-Usos com a JaBUTi	p. 92
6.2	Conclusão	p. 103
7	Conclusão e Trabalhos Futuros	p. 104
	Referências	p. 106
	Apêndice A	p. 108
	Apêndice B	p. 112

Lista de Figuras

1	Importância do uso de Técnicas e Critérios	p. 21
2	Código exemplo para criação de grafo	p. 24
3	Exemplo de grafo de bloco	p. 24
4	Representação de comandos until e case.	p. 25
5	cobertura do critério todos-caminhos.	p. 29
6	exemplo de definição, p-uso e c-uso	p. 31
7	grafo de bloco para fluxo de dados.	p. 32
8	Relação de Inclusão	p. 34
9	Relação de Inclusão entre Critérios de Fluxo de Dados Factível	p. 34
10	Código Exemplo de Elemento Requerido X Peso	p. 37
11	Grafo Exemplo de Elemento Requerido X Peso	p. 37
12	Construção de SBDT - Grafo de Bloco.	p. 38
13	Construção de SBDT - <i>Pre-Dominator Tree</i>	p. 39
14	Construção de SBDT - <i>Post-Dominator Tree</i>	p. 39
15	Construção de SBDT - <i>Basic Block Dominator Tree</i>	p. 40
16	Construção de SBDT - <i>Super Block Dominator Tree</i>	p. 41
17	Instruções <i>IF</i> , <i>IF-ELSE</i> e <i>CASE</i>	p. 43
18	Instruções <i>WHILE</i> , <i>FOR</i> e <i>IF-ELSE</i> dentro de <i>DO-WHILE</i>	p. 43

19	Visualização e Elementos Requeridos para todos-arcos	p.44
20	Visualização e Elementos Requeridos para todos-usos	p.44
21	Lista de Elementos Requeridos	p.45
22	Análise do Código	p.46
23	Lista de variáveis definidas vista pelo código	p.47
24	Análise Inicial do Método Exemplo()	p.49
25	Critério Todos-Nós:Resultado do Primeiro Caso de Teste	p.50
26	Critério Todos-Arcos:Resultado do Primeiro Caso de Teste	p.51
27	Saída Obtida e Esperada	p.51
28	Slicing Tool: Escolha de Elementos Requeridos	p.53
29	Slicing Tool: Local Provável do Defeito	p.54
30	exemplo de código para criação de grafo para ABDR	p.66
31	grafo para aplicação ABDR	p.67
32	população da tabela cliente	p.67
33	método java usado para realização de testes de ABDR.	p.68
34	grafo do código da Figura 33	p.69
35	relação de inclusão dos critérios de ABDR para caminhos executáveis. Fonte: (, 2000)	p.69
36	relação de inclusão dos critérios de ABDR para caminhos não executáveis. Fonte: (, 2000)	p.69
37	Menu de Critérios	p.72
38	Estados da JVM	p.74
39	Definição e T-Uso	p.75

40	Nó SQL Indeterminado (nome do nó em cinza)	p. 76
41	Nó SQL Indeterminado (Entrada Manual de Expressão SQL)	p. 76
42	Grafo Atualizado Após Entrada Manual de Expressão SQL	p. 77
43	Grafo de Fluxo de Dados de ABDR (Nós commit)	p. 79
44	Todos-T-Usos-Ei e Todos-T-Usos-Ed	p. 80
45	Todos-T-Usos-Ed (t-uso dentro de bloco de exceção)	p. 81
46	Distribuição de Pesos Representada em Definições	p. 82
47	Distribuição de Pesos Representada em T-Usos	p. 83
48	Representações de Cobertura de Elemento Requerido	p. 85
49	Interface de Elementos Requeridos	p. 86
50	Análise de Cobertura por Método	p. 86
51	Análise de Cobertura por Critério	p. 87
52	Tela Principal do Sistema de Cosméticos	p. 90
53	Tela de Cadastramento de Funcionário	p. 90
54	Lista de Funcionários Cadastrados	p. 91
55	Dados do Funcionário a Alterar	p. 91
56	Tela de Escolha da Classe Principal	p. 92
57	Tela de Escolha das Classes a se Testar	p. 92
58	Grafo do Método <i>evento_onClick_jButtonCadastrar()</i>	p. 93
59	Constante SQL Analisada pela JaBUTi	p. 94
60	Dados da Análise da Expressão SQL	p. 94
61	Variável Contendo Expressão SQL (Não Analisável pela JaBUTi)	p. 95

62	Inserção Manual de Expressão SQL	p.96
63	Análise da JaBUTi sobre Expressão SQL Inserida Manualmente	p.96
64	Elementos Requeridos do Método de Teste	p.97
65	Inserção de Funcionário Concluída com Êxito	p.98
66	Primeiro Caso de Teste	p.98
67	Alteração do Grafo Após Primeiro Caso de Teste	p.98
68	Resultado Diferente do Esperado	p.99
69	Grafo com Arestas Dependentes de Exceção	p.100
70	Alteração com Dado Inválido	p.101
71	Inserção com Dado Inválido	p.101
72	Resultado de Cobertura dos Elementos Requeridos	p.102
73	Sumário da Análise de Cobertura por Critério	p.102

Lista de Tabelas

1	Casos de Teste para programa da Figura 3	p. 27
2	Associações e caminhos do código da Figura 7	p. 33
3	definições e t-usos do código da Figura 33	p. 59
4	pares definição-t-uso	p. 60
5	dtu-caminhos do código da Figura 33	p. 61
6	dtu-caminhos exercitados com o critério todos-t-usos	p. 62

1 *Introdução*

É conhecida, no meio acadêmico e empresarial, a importância da pesquisa na área de teste de software. A realização de testes visa a aumentar a confiabilidade, melhorar a produtividade (CHUSHO, 1987) e avaliar o quanto o software atende às necessidades especificadas (MYERS, 1979).

Durante o processo de desenvolvimento de software, quanto mais cedo a detecção de defeitos, menos custosa se torna a sua correção. Perry (1995) cita um estudo realizado pela IBM que mostra o custo, em unidades, de se corrigir um erro, podendo ser sintetizado da seguinte forma: defeitos encontrados na fase de requisitos e *design* custam o valor associado à sua própria correção. Já, durante a fase de teste do sistema, após a codificação ter sido implementada, o valor é multiplicado por 10. E multiplicado por 100, quando a correção é feita a partir do momento em que o software foi levado para a fase de produção.

Mas a realização de testes não é uma atividade trivial. Pressman (2001) cita que a atividade de testes pode chegar até a 50% do tempo e custo de um produto de software. Para diminuir o tempo e aumentar a confiabilidade do teste, existem ferramentas que automatizam parte do processo envolvido, como a JaBUTi (VINCENZI et al., 2003), *Java Bytecode Understanding and Testing*, que fornece um conjunto de informações sobre o código Java a ser testado, informações relacionadas à técnica e critérios de teste suportados, e um conjunto de funcionalidades para o auxílio na compreensão do código, visualização dos efeitos dos testes executados e localização de defeitos.

Existem três técnicas de teste de software que possuem características que se complementam e focam diferentes aspectos do software. Estas técnicas são: funcional, estrutural e baseada em defeitos.

A técnica funcional ignora questões de estrutura da programação e se concentra nas especificações funcionais do sistema (BEIZER, 1990). As técnicas de testes estruturais apóiam-se em informações derivadas diretamente da implementação do código e visam a

caracterizar um conjunto de componentes elementares do software que deve ser exercitado pelos testes (SPOTO, 2000). E as técnicas baseadas em defeito utilizam informações sobre os defeitos típicos de desenvolvimento para derivar os requisitos de teste (MORELL, 1990). Para cada técnica, existem diversos critérios de seleção de dados de teste que fornecem os requisitos que estes dados devem cumprir.

Além das técnicas e critérios, os testes são aplicados conforme o escopo a ser considerado. Os testes de unidade são aplicados à menor parte de um programa (módulo), como uma função ou procedimento. Testes de integração consideram a relação entre os módulos. E teste de sistema engloba a aplicação como um todo.

É possível visualizar que os testes devem ir além do programa. Um software pode apresentar defeitos no banco de dados, de forma que este deve ser testado. Spoto (2000) criou um conjunto de critérios de teste para Aplicações de Banco de Dados Relacional (ABDR) baseados na técnica estrutural. Com estes critérios é possível testar a aplicação e sua integração com o banco de dados, dando-se atenção especial à características intrínsecas como o comportamento da aplicação em relação à ação executada na base de dados.

Esta dissertação descreve a implementação do critério de ABDR, todos-t-usos, cujo escopo são os testes de unidade em programas que fazem uso de banco de dados, na Ferramenta de teste JaBUTi.

Apesar das pesquisas na área de testes terem evoluído quantitativamente e qualitativamente, existem lacunas que dificultam a aplicação prática dos métodos, técnicas e critérios conhecidos. Dentre estas, o fato de que os testes se tornam mais complexos a medida em que se deseja aumentar o seu poder em detectar defeitos. Parte deste problema pode ser superado pela automatização de determinados processos, através de ferramentas de teste. Outra questão é a escassez de estudos voltados para aplicações de banco de Dados Relacional e conseqüente dificuldade de utilização prática.

Esta dissertação visa a contribuir para a diminuição destas lacunas ao atribuir automatização ao critério de aplicação de bancos de dados, todos-t-usos, e ao tornar a Ferramenta JaBUTi um instrumento mais completo para realização de testes.

1.2 Objetivo

Esta dissertação tem como objetivo integrar o critério de teste de Aplicações de Banco de Dados Relacional de Spoto (2000), **todos-t-usos**, à Ferramenta de teste JaBUTi, para utilização em programas na linguagem Java.

1.3 Organização

No Capítulo 2, são abordados os principais conceitos relacionados à fase de teste, com ênfase na técnica estrutural. No Capítulo 3, é apresentada a Ferramenta JaBUTi, principais conceitos relacionados e exemplo de aplicação prática. No Capítulo 4, são apresentados conceitos de aplicações de bancos de dados, assim como a definição e análise de inclusão dos critérios todos-t-uso e todos-dtu-caminhos. No Capítulo 5, são discutidas as características funcionais e aspectos de implementação relacionados à inclusão do critério todos-t-usos na JaBUTi. No Capítulo 6, é apresentado um exemplo prático do uso da JaBUTi ao se aplicar o critério implementado. Por fim, o Capítulo 7 apresenta a conclusão.

2 *Teste de Software*

Neste capítulo, são apresentados conceitos de teste de software, abordando-se as definições de defeito, falha, erro, teste de unidade, de integração e de sistema, assim como as definições das técnicas de teste funcional, estrutural e baseada em defeitos. É, também, ressaltada a importância da fase de teste através de exemplo, mostrando-se a relevância da aplicação de critérios na execução dos casos de teste.

2.1 Conceitos Básicos

Ao se testar um software, deve-se ter em mente que o objetivo principal é encontrar defeitos inseridos no código durante sua criação, como citado em Vincenzi (2004): “atividade de teste consiste em uma atividade dinâmica que visa a revelar defeitos”. Inicialmente, então, faz-se necessário conhecer o significado de conceitos como “defeito” e outros diretamente relacionados.

Segundo IEEE (1990), defeito (*fault*) em um programa é um passo, processo ou definição de dados incorreto. Delamaro (1997) cita como exemplo, uma instrução ou comando incorreto. Erro (*error*) é a diferença entre um valor corrente e o valor esperado. Engano (*mistake*) é uma ação humana que produz um resultado incorreto. Falha (*failure*) é o evento notável no qual o programa viola suas especificações.

Para que um defeito seja detectado na fase de teste, faz-se importante adotar técnicas e critérios a serem utilizados, pois o simples ato de executar o programa e comparar os resultados obtidos com os resultados esperados pode dar a falsa impressão de que o objeto testado está de acordo com o especificado.

Por exemplo, no trecho de código mostrado na Figura 1, supondo-se que os valores de `entrada1` e `entrada2` possam variar na faixa de milhões, durante a execução do programa, é provável que não se digite um número entre 11120 e 11150, se as entradas

forem digitadas *ad hoc*. Se houvesse um defeito dentro do laço, como por exemplo, se o programador esquecesse de digitar a linha `entrada1=entrada1 + 1` (um engano), este provavelmente não seria detectado nesta condição de entrada de dados, pois o defeito só seria detectado se o valor digitado estivesse na faixa de 11121 a 11149.

```
scanf ("%d",entrada1);
scanf ("%d",entrada2);
while (entrada1>11120 && entrada1<11150)
{
    entrada2=entrada2 * 2;
    entrada1=entrada1 + 1;
}
```

Figura 1: Importância do uso de Técnicas e Critérios

Conceitos fundamentais relativos a teste de software são domínio de entrada, dados de entrada, saída obtida, saída esperada, caso de teste e conjunto de teste, conforme citado em Spoto (2000) e discutidos a seguir.

Domínio de entrada é o conjunto de todos os dados de entrada (FRANKL; WEYUKER, 1993), também chamados dados de teste, que podem ser aplicados ao programa. **Saída obtida** é o dado mostrado, resultante da execução do programa e **saída esperada** é aquela que deveria ser produzida ao se executar o programa, de acordo com determinado dado de entrada. Um **caso de teste** é formado pelo dado de entrada e dado de saída esperado. **Conjunto de teste** é o conjunto de todos os casos de teste utilizados durante o teste de um software.

Desta forma, observa-se que ao se realizar um teste, é preciso escolher casos de teste e executar o programa para se comparar a saída obtida com a saída esperada. Para isto, faz-se necessário conhecer previamente qual o resultado esperado. Este conhecimento é obtido através de um **oráculo**, que identifica se a saída produzida pela execução de um programa para um determinado dado de teste é a esperada ou não (DELAMARO, 1997). Geralmente é o próprio testador que a partir dos requisitos do software e do estudo do programa, ou trecho de programa, consegue determinar previamente qual a saída esperada para determinado dado de entrada.

Conforme Goodenough e Gerhart (1975), não se pode concluir que um programa esteja correto a partir da execução bem sucedida de uma seleção de dados de teste. Por isso, é importante que se determinem conjuntos de casos de teste com entradas de dados

que abranjam um número significativo de situações em que um defeito possa ser detectado.

O ideal, sob o ponto de vista de detecção de defeitos, seria que todos os possíveis valores de dados de entrada e suas combinações pudessem ser inseridos na fase de teste. Mas este quadro é inviável, na maioria das vezes, dada a alta quantidade de valores e combinações possíveis. Então, busca-se particionar os dados de entradas em subdomínios para que se escolha um representante de cada subdomínio como dado de entrada dos casos de teste. Hamlet e Taylor (1990) relacionam a confiança em um programa com a identificação de subdomínios que possuam alta probabilidade de identificarem a presença de defeitos.

Para se escolher casos de teste existem os métodos de seleção de casos de teste. Para cada método, existe um critério de adequação de casos de teste que é um predicado para avaliar o conjunto de casos de teste para o teste de um programa. Por exemplo, para um método de seleção de casos de teste M , define-se um critério C_M que estabelece: um conjunto de casos de teste é adequado se foi selecionado de acordo com M (DELAMARO, 1997).

Os testes podem ser aplicados levando-se em consideração os procedimentos separadamente (testes de unidade), ou a um procedimento e demais procedimentos relacionados (teste de integração), ou ao sistema como um todo (teste de sistema).

Existem três técnicas para o teste de software: funcional, estrutural e baseado em defeitos.

Técnicas de testes funcionais são aquelas em que se escolhe os requisitos de teste pela especificação funcional (BEIZER, 1990). Esta técnica também é chamada de “caixa preta” (NTAFOS, 1984) e o programa ou módulo é visto como um componente fechado, em que não se utiliza informações sobre a implementação, daí o nome dado.

Técnicas de testes estruturais apóiam-se em informações derivadas diretamente da implementação do código e visam caracterizar um conjunto de componentes elementares do software que deve ser exercitado pelos testes (SPOTO, 2000). Esta técnica é o enfoque deste trabalho.

Técnicas baseadas em defeito utilizam informações sobre os defeitos típicos de desenvolvimento para derivar os requisitos de teste (MORELL, 1990).

2.2 Técnica de Teste Estrutural

A técnica de teste estrutural é também conhecida como teste de “caixa branca” e utiliza a estrutura interna do programa para derivar os dados de teste (GHEZZI; JAZAYERI; MANDRIOLI, 1991). Desta forma, o ponto de partida para esta técnica é a compreensão do código, tarefa que muitas vezes pode ser de alto grau de dificuldade, principalmente se a equipe de teste não for a mesma que desenvolveu o programa.

Para se ter uma visão mais clara sobre o código, utiliza-se como recurso a criação de grafos de programa que são representações gráficas padronizadas que permitem a análise sobre o fluxo de controle do código.

Um grafo possui dois conjuntos de elementos: os nós e os arcos (ou arestas).

Um nó representa um comando ou conjunto de comandos sequenciais, e um arco, o fluxo de controle do programa. Mais detalhes sobre os conceitos relacionados aos grafos são encontrados em Rapps e Weyuker (1985).

Além dos grafos fornecerem uma visão mais clara do código, são normalmente utilizados como base para os critérios de teste estrutural. Por exemplo, o critério todos-arcos (RAPPS; WEYUKER, 1985) requer que todos os arcos do grafo, durante a execução dos casos de teste, sejam exercitados. Os critérios são abordados mais detalhadamente nas Seções 2.2.2 e 2.2.3, sobre critérios de teste de fluxo de controle e de fluxo de dados, respectivamente.

Como exemplo da criação de um grafo, é utilizado o código fonte Java da Figura 2. Será adotado que o nome de cada nó representa o número da linha do primeiro comando do nó ao qual pertence. O grafo do método `Exemplo()` pode ser visto na Figura 3. Os comandos de 0 a 5 são agrupados em um único nó, pois seguem um único fluxo de comandos, sendo a linha 5 o marco de separação de nós por ser um comando condicional (*if* com presença de *else*), o que acarreta em mais de um sucessor (nós 6 e 8). O nó 10 possui mais de um predecessor e possui uma condição (do laço *while*). Os comandos 11 e 12, na Figura 2 pertencem ao nó 11 (Figura 3). Apesar de não haver uma instrução visível, existe um salto do último comando deste nó para o nó 10, isto é, após executado este bloco, a condição em *while* será testada novamente.

```

public void Exemplo (int a)
{
    0   int b=20, c=10;
    1   a=a*2;

    2   System.out.println ("Valor de a=" + a);
    3   System.out.println ("Valor de b=" + b);
    4   System.out.println ("Valor de c=" + c);

    5   if (a>c)
    6   {
    7       c=b;
    8       b=1500000000;
    9   }
    10  else
    11  {
    12      b=20000000001;
    13      c=c+10;
    14  }

    15  while (a>160 && a<200)
    16  {
    17      a=a+1;
    18      b=20000000001;
    19  }

    20  if (b>2000000000)
    21  {
    22      b=1000000000;
    23      b=b*2;
    24      System.out.println ("Valor de a=" + a);
    25      System.out.println ("Valor de b=" + b);
    26      System.out.println ("Valor de c=" + c);
    27  }
}

```

Figura 2: Código exemplo para criação de grafo

Observa-se que o grafo apresentado representa o código-fonte Java. A representação do bytecode seria ligeiramente diferente porque ocorrem duas comparações na linha 10.

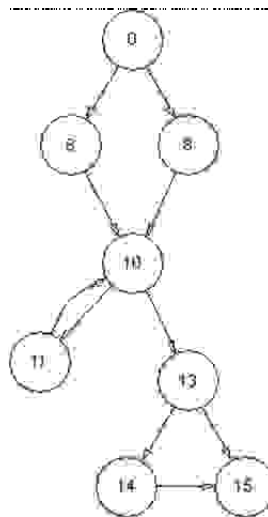


Figura 3: Exemplo de grafo de bloco

O nó 13 é composto pelo comando de mesmo número e representa uma condição *if* sem a presença de *else*. O nó 14 representa o comando que sucede o nó 13 caso a condição seja verdadeira e o nó 15 representa o conjunto de comandos que sucedem o nó 14 (caso a condição em 13 seja verdadeira) ou o nó 13 (caso a condição seja falsa).

Além da representação em grafo dos comandos *if-else* e *while*, Pressman (2001) apresenta modelo de outros dois comandos: *until* e *case* (Figura 4).

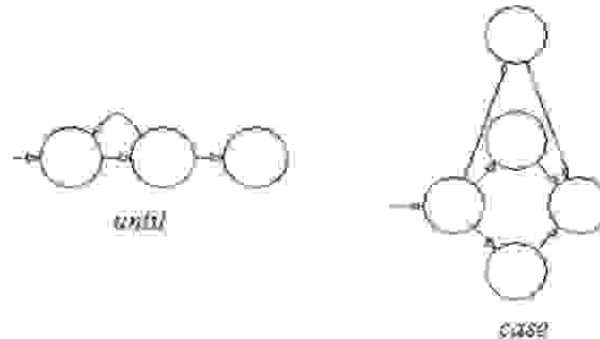


Figura 4: Representação de comandos until e case

Há uma sutil diferença entre a representação de *while* e *until*, o fluxo de saída. Em *while*, o fluxo de saída do laço está localizado em um primeiro nó de forma que existe a possibilidade dos comandos contidos no bloco deste laço (nó 11 da Figura 3) nunca serem executados. Já no comando *until*, o fluxo de saída do laço está no próprio bloco contido no laço, indicando que este obrigatoriamente será executado pelo menos uma vez.

Tendo-se o conhecimento sobre como interpretar o grafo, torna-se mais fácil analisar o código, pois é possível identificar o fluxo de controle a partir de um nível mais alto, abstrato, e detectar a presença de comandos ou conjuntos de comandos característicos e freqüentemente utilizados, através das disposições dos nós e arcos, como demonstrado com os comandos *if-else*, *while*, *until* e *case*.

Após o estudo realizado sobre o código e seu entendimento, deve-se escolher os critérios de teste que serão considerados na criação dos casos de teste. O item seguinte apresentará alguns dos principais critérios de fluxo de controle e de dados.

2.2.2 Critérios de teste baseados em fluxo de controle

Os critérios baseados em fluxo de controle têm como foco principal os comandos do código. Como fluxo deve-se entender o curso que será tomado entre um comando e outro, durante uma execução.

Existem vários critérios baseados em fluxo de controle. Dentre os mais conhecidos estão: *todos-nós*, *todos-arcos* e *todos-caminhos* (RAPPS; WEYUKER, 1982).

- Critério **todos-nós**: requer que cada comando do programa seja executado pelo menos uma vez.
- Critério **todos-arcos**: requer que toda transferência de controle seja exercitada pelo menos uma vez.
- Critério **todos-caminhos**: requer que todos os caminhos sejam exercitados pelo menos uma vez. Um caminho é uma sequência finita de nós $(N_1, \dots, N_k)$, $k \geq 2$, tal que, para todo nó N_i , $1 \leq i \leq k - 1$, existe um arco (N_i, N_{i+1}) para $i = 1$ até $k - 1$.

Para o código da Figura 2, ao se adotar o critério todos-nós, é requerido que todos os comandos do método *Exemplo()* sejam executados pelo menos uma vez, isto é, todos os nós devem ser exercitados. Assim, deve-se criar casos de teste que cumpram esta exigência. Observando o grafo, é possível notar que não há como exercitar todos os nós em apenas uma execução, pois a primeira condição do programa (nó 0) faz com que o fluxo se divida em dois caminhos possíveis e independentes. Desta forma, se a condição $a > b$ for verdadeira, o nó 6 será exercitado e o 8, não. Da mesma maneira, se a condição $a > b$ for falsa, o nó 8 será exercitado e o 6, não.

Deve-se observar uma importante diferença entre as condições *if-else* e *if* em relação ao fluxo de controle. Como mostrado, a condição *if-else* implica que durante uma execução, um nó sucessor não será exercitado. No caso de uma condição *if*, esta situação não necessariamente ocorre. Por exemplo, se a condição *if* do nó 13 for verdadeira, todos os nós sucessores a este nó serão exercitados, como mostra o fluxo de controle do nó 14 para o 15, na Figura 3.

Aplicando-se o critério todos-nós, seria possível descobrir um defeito como a falta da linha 11, através da ocorrência de um *loop* infinito. Supondo-se que os casos de teste se baseassem em entradas de dados inseridas aleatoriamente, é bastante provável que não se digitasse um número correspondente à condição de entrada do laço.

Para que o critério todos-nós seja satisfeito, no exemplo dado, pode-se criar dois casos de testes cujos valores do parâmetro sejam:

- 1 $a = 3$ (ou qualquer valor inteiro menor que 5). Este caso de teste exercitará os nós 0, 8, 10, 13, 14 e 15.

II $a = 90$ (ou qualquer valor inteiro maior que 80 e menor que 100). Este caso de teste exercitará os nós 0, 6, 10, 11, 13, 14 e 15.

Tendo-se exercitado os nós 0 (em I e II), 6 (em II), 8 (em I), 10 (em I e II), 11 (em II), 13 (em I e II), 14 (em I e II) e 15 (em I e II), todos os nós foram exercitados. Os resultados das execuções apresentados na tela podem ser vistos na Tabela 1.

O critério todos-arcos requer que todas as arestas sejam exercitadas pelo menos uma vez. Utilizando-se os dois casos de teste anteriores, nota-se que apesar de se terem exercitado todos os nós, o arco do nó 13 ao 15, representado como (13,15) não foi exercitado em nenhuma das execuções, pois em ambos o valor de b , no momento em que a condição da linha 13 é verificada, é sempre maior que 2.000.000.000.

Tabela 1: Casos de Teste para programa da Figura 3

Nó	Variável	Caso de Teste 1	Caso de Teste 2	Caso de Teste 3
0	a	6	180	300
	b	20	20	20
	c	10	10	10
15	a	6	200	300
	b	2.000.000.000	2.000.000.000	-1.294.967.296
	c	20	20	20

Assim, o critério todos-arcos requer mais dados de teste para que seja satisfeito em relação ao critério todos-nós. Faz-se então, necessário que se realize pelo menos mais um caso de teste em que o valor de b no momento em que a linha 13 for exercitada, seja menor ou igual a 2.000.000.000.

Um novo caso de teste deve exercitar um caminho em que a seja maior que c na linha 5, mas que não passe pelo bloco do laço *while*, por exemplo:

III $a = 150$ (ou qualquer valor em que $6 \leq a \leq 80$ e $100 \leq a \leq 2^{31} - 1$). Este caso de teste exercitará os nós 0, 6, 10, 13 e 15.

Nesta execução poderá ser detectado que o valor da multiplicação de b na linha 15 será maior do que o limite suportado pelos números inteiros em Java (valor limite máximo de $2^{31} - 1$, ou 2.147.483.647), ocasionando um valor incorreto (Tabela 1, em vermelho), defeito não previsto antes.

O valor em vermelho mostrado na Tabela 1 deveria representar o resultado da multiplicação de 1.500.000.000 por 2, isto é, 3.000.000.000.

O critério todos-caminhos requer que todos os caminhos possíveis, do nó 0 ao nó 15 sejam exercitados. Mas este critério pode não ser praticável, principalmente na presença de laço. Com o código utilizado nos casos de teste, pode-se derivar os seguintes caminhos:

- $\langle 0, 6, 10, 11, 10, 13, 14, 15 \rangle$
- $\langle 0, 6, 10, 13, 15 \rangle$
- $\langle 0, 6, 10, 11, 10, 11, 10, 13, 14, 15 \rangle$
- $\langle 0, 8, 10, 13, 14, 15 \rangle$

Porém, diversos outros caminhos podem ser derivados. Focando-se apenas no laço, pode-se gerar 19 caminhos inserindo-se valores iniciais de a entre 81 e 99 (adotando-se estes valores, na linha 10 tornar-se-ão, respectivamente, 162 e 198), como os quatro caminhos, a seguir:

- $\langle \dots, 10, 11, 10, 11, 10, 13, \dots \rangle$ para $a = 99$
- $\langle \dots, 10, 11, 10, 11, 10, 11, 10, 11, 10, 13, \dots \rangle$ para $a = 98$
- $\langle \dots, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 13, \dots \rangle$ para $a = 97$
- $\langle \dots, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 13, \dots \rangle$ para $a = 96$

Um conceito importante a ser citado é o de caminhos factíveis ou executáveis. Observando-se o grafo, pode-se idealizar o caminho $\langle 0, 8, 10, 11, 13, 15 \rangle$.

Mas ao observar o código, nota-se que não é possível exercitar o nó 8 e entrar no laço *while*. Isto porque, para a primeira situação, a deve ser menor ou igual a c , cujo valor é 10. Mas se $a \leq 10$, não é possível entrar no *loop*, pois a deve ser maior que 160 e menor que 200. Com esta impossibilidade, diz-se que o caminho idealizado é infactível.

Outro ponto a ser verificado é a complexidade do teste. Em um código cujo laço possa ser percorrido de 1 a 100 vezes, por exemplo, haverá 100 caminhos possíveis apenas

levando-se em consideração o laço. Se este código possuir, por exemplo, um comando *case* com cinco valores possíveis para a variável em questão e ao final, todos os caminhos sejam possíveis de serem combinados com o laço, haverá 500 caminhos possíveis. Dada a quantidade de casos de teste para que cada caminho seja coberto, adotar este critério para códigos com laço pode ser inviável.

Porém, há de se notar que o critério todos-caminhos pode detectar erros que não seriam identificáveis pelo critério todos-arcos (e conseqüentemente, todos-nós). Ghezzi, Jazayeri e Mandrioli (1991) demonstram um exemplo, utilizando o código da Figura 5.

```
if x≠0 then
    y:=5;
else
    z:=z - x;
end if;
if z>1 then
    z:=z/x;
else
    z:=0;
end if;
```

Figura 5: cobertura do critério todos-caminhos.

Realizando-se dois casos de teste em que os valores iniciais de x e z sejam respectivamente:

I $x = 0, z = 1$;

II $x = 1, z = 3$.

Todos os arcos serão cobertos e nenhum erro será detectado.

Para que o critério todos-caminhos seja exercitado, dois novos casos de teste se fazem necessários, por exemplo:

III $x = 0, z = 3$;

IV $x = 1, z = 1$.

O caso de teste III levará a uma divisão por zero, não prevista com o critério todos-arcos.

2.2.3 Critérios de teste baseados em fluxo de dados

Até o momento, os critérios abordados levaram em consideração apenas o fluxo de controle. Porém, o fluxo das próprias variáveis, ou dados, não foram avaliados.

Existem diversos critérios baseados em fluxo de dados, dentre eles os pertencentes à Família de Critérios Potenciais Usos (FCPU), de Maldonado (MALDONADO, 1991), e à Família de Critérios Baseados em Fluxo de Dados (FCFD), de Rapps e Weyuker (RAPPS; WEYUKER, 1982) e (RAPPS; WEYUKER, 1985).

Dois critérios pertencentes à FCFD se relacionam diretamente com os critérios de unidade de Spoto (2000) para aplicações de Banco de Dados Relacional, e com o entendimento do uso da Ferramenta JaBUTi, sendo eles: *todos-usos* e *todos-du-caminhos*.

A estes critérios, existem cinco conceitos fundamentais associados: *definição*, *uso*, *caminho livre de definição*, *caminho livre de laço* e *du-caminho*.

Definição e *uso* são ações que podem ocorrer a uma variável. Uma definição de uma variável ocorre quando um valor é armazenado em uma posição de memória e esta ação é executada basicamente de três formas possíveis:

- Quando a variável está no lado esquerdo do sinal de atribuição. Em $x := a + 1$, há uma definição da variável x ;
- Em um comando de entrada, por exemplo, $\text{readln}(x)$;
- Em chamadas de procedimento como parâmetro de saída.

A ação de *uso* de uma variável ocorre quando se recupera um valor em uma posição de memória associada a esta variável (FRANKL; WEYUKER, 1988) e pode ser dividido em duas categorias: *p-uso*, ou uso predicativo, e *c-uso*, ou uso computacional.

O *p-uso* ocorre quando esta ação afeta o fluxo de controle, como em uma condição *if* ou condição de laço. O *c-uso* afeta diretamente uma computação, como em uma atribuição ou leitura de entrada de dados.

Seja o trecho de código da Figura 6 e seu respectivo grafo, verifica-se que o nó 1 possui duas definições e dois usos. A variável x possui uma definição na linha 1 e um c-uso na linha 2. A variável y possui uma definição na linha 2 e um p-uso na linha 3.

Há duas definições de z (linha 1 do nó 1 e linha 6 do nó 6) e três c-usos (linha 2 do nó 1, linha 6 do nó 6 e linha 7 do nó 6). Apesar de ocorrer uma definição na linha 6, primeiro há um c-uso de z , e por isso diz-se que existe um caminho livre de definição do nó 1 ao 6 para o uso de z da linha 6. Porém, tal definição ocorre antes do c-uso da linha 7, significando que não há um caminho livre de definição do nó 1 ao 6 para o uso de z da linha 7.

```

1 readln (z);
2 x:=z+1;
3 if (x>10) then
4   writeln('X é maior que 10. X=',x)
5   else
6     writeln('X é menor ou igual a 10. X=',x);
7 z:=z + 1;
8 writeln ('Valor de z=',z);

```

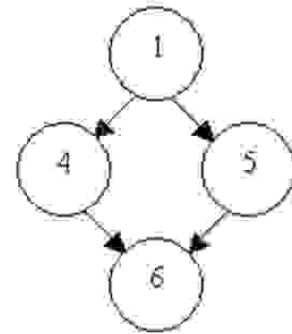


Figura 6: exemplo de definição, p-uso e c-uso

Um *c-uso* está associado a um nó. Já um *p-uso*, por influenciar o fluxo de controle, está associado ao arco. No exemplo, diz-se que existem p-usos de x nos arcos (1,4) e (1,5).

Um caminho $(i, n_1, \dots, n_m, j)$, $m \geq 0$ com uma definição da variável x no nó i e que não contenha nenhuma redefinição de x nos nós $n_1, \dots, n_m$ é chamado de *caminho livre de definição* com respeito a (c.r.a.) x do nó i ao nó j e do nó i ao arco (n_m, j) . Neste caso, pode existir uma redefinição de x no nó j (SPOTO, 2000).

Desta forma, existem quatro pares definição-uso, sendo dois pares relacionados ao p-uso da linha 3, cujos arcos são (1,4) e (1,5) e dois pares relacionados aos c-usos das linhas 4 e 5.

Um caminho é dito *livre de laço* se todos os nós pertencentes a ele são distintos, (RAPPS; WEYUKER, 1985) e é classificado como caminho simples se todos os nós forem distintos, exceto, possivelmente, o primeiro e último. E por fim, um *du-caminho* é um caminho $(n_1, \dots, n_j, n_k)$ cujo par definição-uso forma um caminho livre de definição e:

- n_k é um c-uso e $(n_1, \dots, n_j, n_k)$ é um caminho simples, ou;
- (n_j, n_k) é um p-uso e $(n_1, \dots, n_j, n_k)$ é um caminho livre de laço.

Com estes conceitos, é possível compreender a definição dos critérios a serem abordados. Segundo Rapps e Weyuker (1982) e (1985):

- Critério **todos-usos**: requer que todas as associações entre cada *definição* de variável e subseqüentes *c-usos* e *p-usos* dessa definição sejam exercitadas pelo menos uma vez;
- Critério **todos-du-caminhos**: requer que todas as associações entre cada *definição* de variável e subseqüentes *p-usos* ou *c-usos* dessa definição sejam exercitadas por todos os caminhos livres de definição e livres de laços que cubram essas associações; ou seja, requer que todos-du-caminhos sejam exercitados.

O grafo de bloco utilizado até o momento é direcionado para fluxo de controle. Para o fluxo de dados pode-se adaptar o anterior, incluindo as definições e usos existentes. O exemplo, a seguir, é uma adaptação do código de Ghezzi, Jazayeri e Mandrioli (1991) da Figura 7.

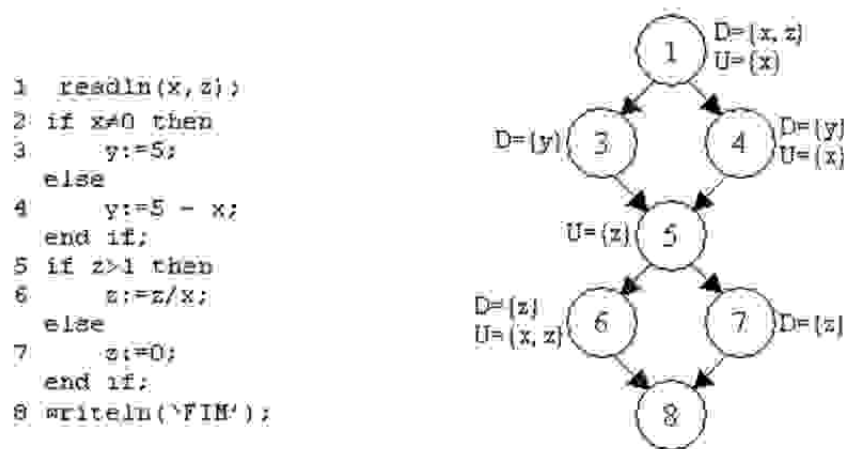


Figura 7: grafo de bloco para fluxo de dados.

Para determinar os casos de teste que satisfaçam os requisitos todos-usos, deve-se inicialmente determinar todas as associações definição-uso: $(1, (1, 3), x)$, $(1, (5, 6), z)$, $(1, (1, 4), x)$, $(1, (5, 7), z)$, $(1, 6, x)$ e $(1, 4, z)$.

Um conjunto de casos de teste possível para que se exercitem todos os requisitos de todos-usos é entrar com os seguintes valores para x e z :

- Caso de teste 1: $x = 0$ e $z = 1$;
- Caso de teste 2: $x \neq 0$ e $z = 2$;

O caso de teste 1 garante que as associações definição-uso $(1, (1,4), x)$, $(1,4,z)$ e $(1, (5,7), z)$ sejam exercitados. O caso de teste 2 garante que as associações $(1, (1,3), x)$, $(1,6,x)$ e $(1, (5,6), z)$ sejam exercitados.

Para o critério **todos-usos** os caminhos que deverão ser exercitados são mostrados na Tabela 2. Observa-se quatro caminhos principais a serem exercitados (em azul), pois os demais estão inclusos nestes.

Tabela 2: Associações e caminhos do código da Figura 7

Associação	Caminho
$(1, (1,3), x)$	$\langle 1, 3 \rangle$
$(1, (1,4), x)$	$\langle 1, 4 \rangle$
$(1,6,x)$	$\langle 1, 3, 5, 6 \rangle$ e $\langle 1, 4, 5, 6 \rangle$
$(1, (5,6), z)$	$\langle 1, 3, 5, 6 \rangle$ e $\langle 1, 4, 5, 6 \rangle$
$(1, (5,7), z)$	$\langle 1, 3, 5, 7 \rangle$ e $\langle 1, 4, 5, 7 \rangle$
$(1,4,z)$	$\langle 1, 4 \rangle$

Os dois casos de teste do critério todos-usos exercitaram os caminhos 1,3,5,6 e 1,4,5,7. Para os demais, pode-se executar os dois casos de teste a seguir:

- Caso de teste 3: $x \neq 0$ e $z = 1$;
- Caso de teste 4: $x = 0$ e $z = 2$.

Com estes casos de teste, a exceção gerada pela divisão por zero é detectada, o que não ocorreu com a execução de casos de teste do critério todos-usos.

2.3 Análise de Inclusão

Segundo Vincenzi (2004), uma relação de inclusão estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. Rapps e Weyuker (1985) cita que, dados dois critérios C e C' , diz-se que C inclui C' se para qualquer programa P , todo conjunto de teste C adequado é também C' adequado. O critério C inclui estritamente o critério C' , denotado por $C \Rightarrow C'$, se C inclui C' e C' não inclui C . Quando nem C inclui C' nem C' inclui C diz-se que os critérios C e C' são incomparáveis.

Em relação aos critérios apresentados neste capítulo, diz-se que, para programas em que todos os elementos requeridos pelos critérios adotados são exercitáveis, o critério

todos-caminhos inclui todos-du-caminhos. Este por sua vez inclui o critério todos-usos que inclui todos-arcos, e que finalmente inclui o critério todos-nós (NTAFOS, 1988). Na Figura 8 é mostrada a relação de inclusão.



Figura 8: Relação de Inclusão

Desta forma, se um conjunto de casos de teste T_1 , executados em um programa P , satisfizer o critério todos-caminhos, também satisfará o critério todos-du-caminho. Se um conjunto de casos de teste T_2 satisfizer o critério todos-usos, também satisfará o critério todos-arcos. E o mesmo conceito é aplicado para o critério todos-arcos em relação ao critério todos-nós.



Figura 9: Relação de Inclusão entre Critérios de Fluxo de Dados Factível

Por outro lado a presença de elementos requeridos não factíveis não permite que se encontre um conjunto de casos de teste que seja adequado ao critério. Se for considerado que um critério de teste C^* é satisfeito se todos os elementos requeridos factíveis forem exercitados, a relação de inclusão é alterada, como segue na Figura 9. A esta relação de inclusão, Frankl e Weyuker (1988) chamam de *relationship among feasible data flow testing criteria* ou relacionamento entre critérios de teste de fluxo de dados factível. Mais

detalhes sobre análise de inclusão podem ser vistos em Frankl e Weyuker (1988), Ntafos (1988) e Zhu (1996).

2.4 Conclusão

Foram apresentados conceitos relacionados aos testes de software. Inicialmente recapitularam-se definições básicas sobre teste, como defeito, falha, erro, teste de unidade, teste de integração, teste de sistema, assim como as principais técnicas envolvidas na fase de testes.

Em seguida apresentou-se mais detalhadamente a técnica estrutural e temas relacionados, como grafos e seus padrões com relação aos comandos comumente utilizados (*if-else*, *case*, e laços); critérios de teste, com exemplos práticos que mostraram diferenças entre eles, exemplos de defeitos que podem ser encontrados, e análise de inclusão.

3 Ferramenta de Teste JaBUTi

Ao se aplicar a técnica de teste estrutural, é preciso conhecer a implementação do código. No Capítulo 2 é mostrado que o conhecimento da estrutura interna do programa pode ser abstraído na forma de um grafo de fluxo de controle e, a partir dele, derivarem-se os casos de teste baseados em determinado critério de teste, como todos-nós e todos-arcos.

Porém, os exemplos mostrados nos capítulos anteriores apresentam trechos de código ou unidades de programa, que representam uma pequena parte do software. Criar um fluxo de controle para cada unidade de programa, assim como detectar todos os elementos requeridos e analisar a cobertura dos casos de teste, pode ser um processo extremamente custoso, se feito por um indivíduo ou grupo, além de se correr o risco de não realizar tais tarefas corretamente. Para diminuir o tempo despendido neste processo e diminuir os riscos, existem ferramentas de teste como a Poke-tool (CHAIM, 1991) e a JaBUTi (VINCENZI et al., 2003).

A JaBUTi, uma ferramenta de teste estrutural, de unidade, automatiza parte do processo de teste, criando os grafos para cada método Java, apresentando todos os elementos requeridos para os critérios todos-nós, todos-arcos, todos-usos e todos-potenciais-usos, e fornecendo um grupo de funcionalidades que permite a compreensão e análise sobre os testes. As análises realizadas pela ferramenta são feitas sobre o bytecode.

Neste capítulo são apresentados conceitos utilizados pela JaBUTi relacionados à análise sobre o código, bem como o processo automatizado fornecido pela ferramenta e algumas de suas principais funções.

3.1 Conceitos Básicos

Quando se executa um caso de teste, é natural que ao se exercitar um determinado elemento requerido, outros elementos requeridos também sejam exercitados. Como

exemplo, será utilizado o trecho de código da Figura 10 e seu respectivo grafo da Figura 11.

```

0  if (a>b)
1      System.out.println("A é maior que B");
    else
2      System.out.println("B é maior ou igual a A");
3  If (c<d)
4      System.out.println("C é maior que D");
5  System.out.println("Valor de A é " + a + " e valor de B é " + b);

```

Figura 10: Código Exemplo de Elemento Requerido X Peso

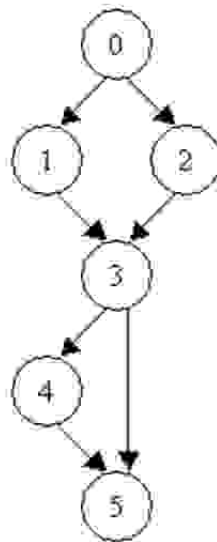


Figura 11: Grafo Exemplo de Elemento Requerido X Peso

Ao se escolher um caso de teste que exercite o nó 3, com certeza três nós (elementos requeridos pelo critério todos-nós) serão exercitados: 0, 3 e 5. Porém, ao se escolher um caso de teste que exercite o nó 4, certamente quatro nós serão exercitados: 0, 3, 4 e 5. Desta forma, diz-se que o elemento requerido “nó 4” possui um peso maior que o elemento requerido “nó 3”. Portanto, quanto maior o peso de um elemento requerido, maior o número de elementos requeridos que certamente serão exercitados caso esse elemento seja exercitado.

A JaBUTi possui um sistema de distribuição de pesos para cada critério suportado. Em todos os casos, o conceito geral de peso é o acima apresentado. A representação dos pesos é apresentada na forma de cores com uma legenda que determina o valor numérico das cores. Assim, para o critério **todos-nós**, cada nó é pintado com sua devida cor, variando de 0 (cor branca, que indica que o elemento já foi exercitado) a um número inteiro maior que zero, sendo o maior peso representado pela cor vermelha, como pode ser visto a partir da Seção 3.2. Para o critério **todos-arcos**, cada nó de condição,

ou nó que ocasiona dois ou mais fluxos de controle, é pintado com uma cor. E para o critério **todos-usos**, cada nó que possui uma definição de uma variável que possua uma associação definição-uso, é pintado com uma cor.

Para computar os pesos dos elementos requeridos, utiliza-se os métodos descritos nas próximas seções.

3.1.1 Distribuição de Peso para o Critério Todos-Nós

Para a distribuição de pesos dos elementos requeridos do critério todos-nós, é utilizado um algoritmo que se baseia no grafo chamado *Super Block Dominator Tree* (SBDT). Este grafo permite que se visualize quais nós serão obrigatoriamente exercitados se um nó x for exercitado e origina-se da fusão de dois outros grafos: *Pré-Dominator Tree* e *Post-Dominator Tree*. Para explicar sua construção, faz-se necessário conhecer os conceitos de **pré-domínio** e **pós-domínio**, que serão explicados a seguir.

Um nó u **pré-domina** um nó v se todo caminho do nó de entrada para v contém u (AGRAWAL, 1994). Seja o grafo da Figura 12, as setas pontilhadas representam o desvio do fluxo de controle para um bloco de tratamento de exceção.

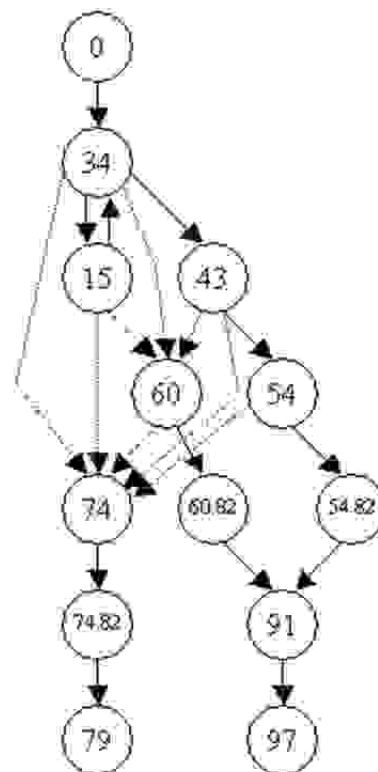


Figura 12: Construção de SBDT - Grafo de Bloco.

Conforme a definição de **pré-domínio**, é possível verificar que o nó 0 pré-domina o nó 34 porque do nó de entrada até o nó 34, necessariamente se passa pelo nó 0 (neste caso, o nó 0 é o próprio nó de entrada). Da mesma forma, o nó 34 pré-domina o nó 15, pois do nó de entrada para o nó 15, obrigatoriamente se passa pelo nó 34. Já o nó 54.82 não pré-domina o nó 91, pois do nó de entrada para o nó 91, não necessariamente se exercita o nó 54.82 (pode-se exercitar o nó 91 através do nó 60.82). Mas o nó 34 pré-domina o nó 91, já que para que o nó 91 seja exercitado obrigatoriamente se exercita o nó 34. Seguindo este conceito, monta-se o grafo *Pre-Dominator Tree* como o da Figura 13 (adaptado de Vincenzi et al. (2003)).

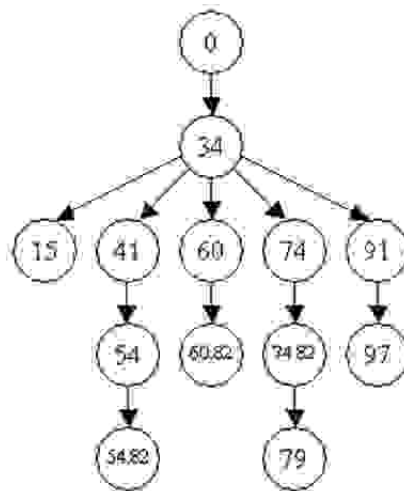


Figura 13: Construção de SBDT - *Pre-Dominator Tree*.

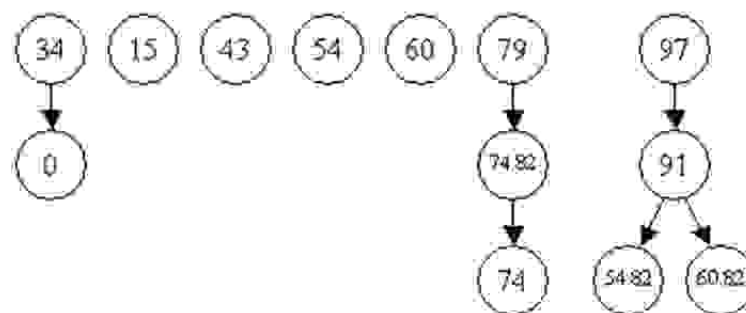


Figura 14: Construção de SBDT - *Post-Dominator Tree*.

Um nó w **pós-domina** (ou domina inversamente) um nó v se todo caminho de v para um nó de saída contém w . Desta forma, conforme o grafo da Figura 14, o nó 91 pós-domina o nó 54.82, pois deste nó para o nó de saída, certamente é exercitado o nó 91. Da mesma forma, o nó 91 pós-domina o nó 60.82. Porém, o nó 91 não pós-domina o nó 34, pois do nó 34 ao nó de saída, não necessariamente se exercita o nó 91, já que é possível sair pelo nó 79.

Nota-se que os nós 15, 43, 54 e 60 não possuem relação de pós-dominância com outros nós. Isto se dá porque não há um caminho que passe por um nó w e um nó de saída e que obrigatoriamente passe por estes nós.

O passo seguinte para a construção do *Super Block Dominator Tree* (Figura 15) é a fusão do grafo *Pre-Dominator Tree* e *Post-Dominator Tree*, que gerarão o grafo *Basic Block Dominator Tree*. Esta fusão consiste em recriar em um único grafo as relações entre cada nó de ambos os grafos citados.

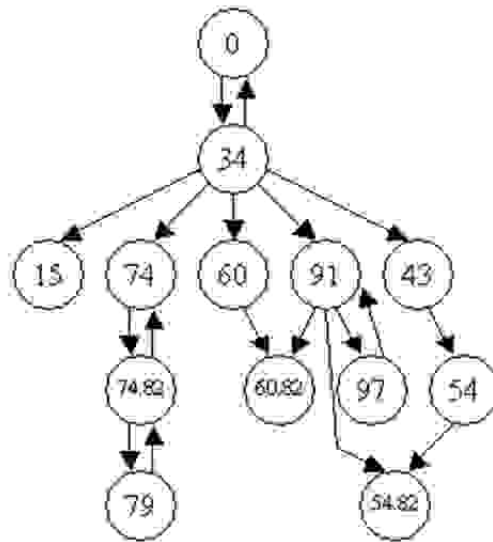


Figura 15: Construção de SBDT - Basic Block Dominator Tree.

Tendo-se criado o *Basic Block Dominator Tree*, o passo final é unir todos os nós que possuem relação de fluxo bidirecional, isto é, se existirem dois nós x e y em que exista uma relação de x para y e de y para x , estes nós devem ser unidos. Com base no grafo da Figura 15, pode-se unir os nós 74 e 74.82, e estes ao nó 79. Une-se, também, os nós 91 e 97, e os nós 0 e 34. O grafo apresentado na Figura 16 ilustra o resultado final, o *Super Block Dominator Tree*.

Com a construção deste grafo, é possível visualizar quais nós serão necessariamente exercitados se um nó x for exercitado. Ao se exercitar o nó 60.82, é possível verificar por este grafo (Figura 16) que os nós 91, 97, 60, 0 e 34 serão obrigatoriamente exercitados, *bastando ver quais são os nós que o precedem*. Todos estes nós pertencem a um super bloco.

Os pesos são distribuídos da seguinte forma: dado o grafo *Super Block Dominator Tree*, é atribuído a cada nó o valor de quantos nós este contém mais a quantidade de nós que o precedem. O primeiro nó possui valor 2, pois contém os nós 0 e 34. O nó 15 possui

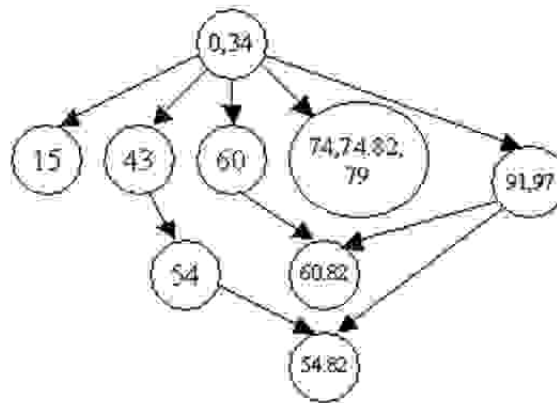


Figura 16: Construção de SBDT - Super Block Dominator Tree.

valor 3, pois contém um nó (15) mais 2 (do nó que o precede - 0, e 34). O nó 60.82 possui valor 6, pois contém um nó (60.82) mais os nós 60, 0, 34, 91 e 97.

Desta maneira, os pesos indicam quantos nós necessariamente serão exercitados caso um nó x seja exercitado.

3.1.2 Distribuição de Peso para o Critério Todos-Arcos

Para se distribuir os pesos, inicialmente é criada uma lista de todos os elementos requeridos para o critério todos-arcos. Destes, apenas os nós que contiverem uma instrução de decisão (como *if* ou *while*) serão pintados. A distribuição é feita separadamente para os fluxos independentes e dependentes de exceção.

Dado um elemento requerido, para cada nó será considerado o conjunto de todos os nós pertencentes ao seu super bloco. Assim, a partir do elemento requerido (34,15), por exemplo, dois conjuntos serão derivados: os nós pertencentes ao super bloco do nó 34 e os nós pertencentes ao super bloco do nó 15. Para o primeiro nó, com base no grafo *Super Block Dominator Tree* da Figura 16, tem-se o conjunto $C_1 = \{0, 34\}$. Para o segundo nó, tem-se o conjunto $C_2 = \{0, 15, 34\}$.

Com estes dados, a ferramenta irá determinar quantos elementos requeridos, conjunto E_1 , existem em que o seu primeiro nó está contido no conjunto C_1 e que o segundo nó está contido em C_2 . O peso do elemento requerido (34,15) será a soma de todos estes elementos requeridos contidos mais 1 (o próprio elemento requerido). Neste exemplo, o peso é 2, pois o elemento requerido (0,34) pertence a E_1 . Isto significa que passando-se por (34,15), obrigatoriamente se exercita o arco (0,34).

3.1.3 Distribuição de Peso para o Critério Todos-Usos

A distribuição de pesos para este critério segue o mesmo conceito de todos-arcs. É gerada uma lista com todos os elementos requeridos para todos-usos, mas são pintados apenas os nós que possuem uma definição pertencente a uma associação definição-uso.

Dado um elemento requerido, são gerados os conjuntos de todos os nós pertencentes ao super bloco do nó de definição (conjunto C_1) e do nó de uso (se o uso for predicativo, um conjunto C_2 para o primeiro nó e um conjunto C_3 para o segundo nó). Determina-se quantos elementos requeridos possuem seus respectivos nós contidos nos conjuntos C_1 , C_2 e C_3 . O peso corresponde à soma destes elementos requeridos mais um (o próprio elemento requerido em questão).

3.2 Compreensão do Código

Para mostrar como a ferramenta é utilizada para se compreender o código, será usado o código mostrado no Apêndice A. Ao se executar o programa, são pedidos três dados de entrada. Um deles, variável *LimZ*, será passado como parâmetro e representa um valor em graus. Será calculado o **seno** deste valor e seu resultado demonstrado em um gráfico senóide. Outros dois, *x* e *z*, serão utilizados para realização de um determinado cálculo. Será considerado que em sua especificação, valores maiores que 360 para *LimZ*, serão substituídos por 360.

3.2.1 Grafo

Além de montar e apresentar os grafos, a ferramenta fornece dados sobre o método representado. A divisão entre um nó e outro se dá basicamente de quatro formas (VIN-CENZI et al., 2003): na presença de uma instrução de salto, quando uma instrução possui mais de um sucessor (como em uma instrução condicional), quando o sucessor tem mais de um predecessor (como a instrução seguinte a um bloco *if-else*) e quando o sucessor de uma instrução não possui o mesmo conjunto de sucessores secundários que seu predecessor.

O programa do exemplo possui um método, *FatorZ()*, sem qualquer funcionalidade, mas com o objetivo de se demonstrar a representação das instruções *if*, *if-else*, *case*, *for*, *while* e *do-while* em grafo (comentadas no Capítulo 2).

Na Figura 17, os nós 0, 8 e 16 representam o comando *if*. Os nós 16, 22, 33 e 41 correspondem ao comando *if-else*. E do nó 41 ao 109, a instrução *case*.

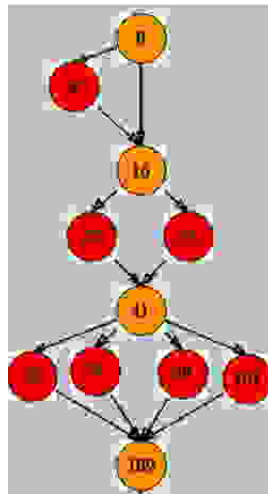


Figura 17: Instruções *IF*, *IF-ELSE* e *CASE*

Na Figura 18, os nós 109, 111 e 116 representam um laço *for*. Os nós 130 e 136 são a representação de uma instrução *while*, e os nós 151, 164, 175 e 183 são a apresentação gráfica de uma instrução *if-else* dentro de um *do-while*.

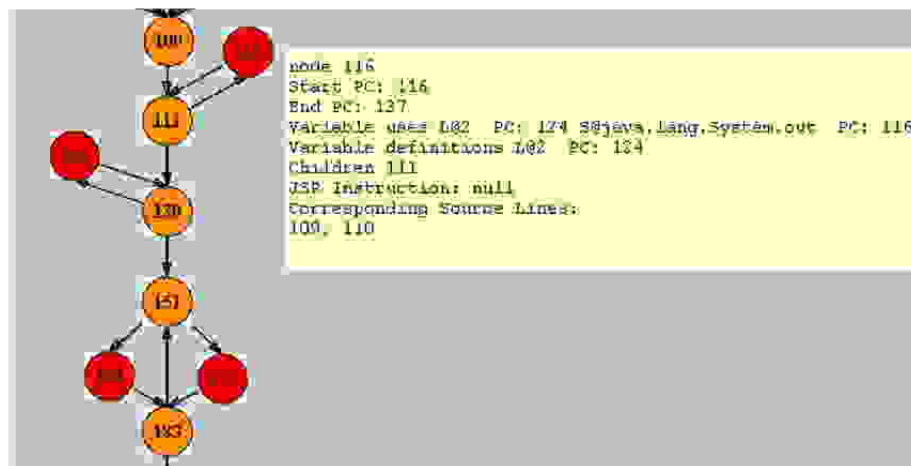


Figura 18: Instruções *WHILE*, *FOR* e *IF-ELSE* dentro de *DO-WHILE*

Dados sobre cada nó são mostrados, como na Figura 18. O campo *node* indica o nome do nó. *Start PC* mostra qual a primeira linha bytecode do nó e *EndPC* a última. A JaBUTi enumera cada método independentemente, começando pela linha 0. Em seguida, *Variable uses* e *Variable definitions* apresentam, respectivamente, quais variáveis estão sendo usadas e quais estão sendo definidas neste nó e a linha correspondente (do bytecode). *Children* mostra quais os nós que precedem o nó atual. *JSP Instruction* relaciona um possível método chamado (no exemplo, nenhum método está sendo chamado por este nó).

Por fim, *Corresponding Source Lines* apresenta quais são as linhas correspondentes no código-fonte (caso esteja disponível).

Para o critério **todos-nós**, todos os elementos requeridos são mostrados de forma bastante evidente. Mas para os demais critérios, como todos-arcos e todos-usos, esta visualização é reforçada. Assim, para o critério **todos-arcos**, são pintados apenas os nós de decisão. Ao clicar com o botão em um destes nós, o de decisão passa a ter cor azul e serão pintados todos os nós filhos (sucessores), como demonstrado na Figura 19.

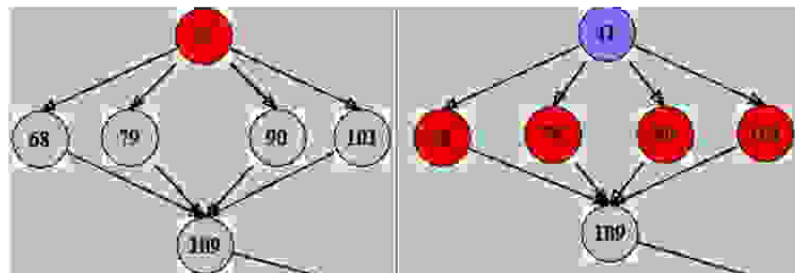


Figura 19: Visualização e Elementos Requeridos para todos-arcos

Já, para o critério **todos-usos** (Figura 20), ao se clicar com o botão direito do mouse sobre um nó, são listadas todas as variáveis cujas definições se encontram neste nó. Ao se escolher uma das variáveis, é pintado de azul o nó de definição, e com outras cores, todos os nós (c-uso) ou arcos (p-uso) que possuem uso e fazem parte da associação com a variável marcada em azul.

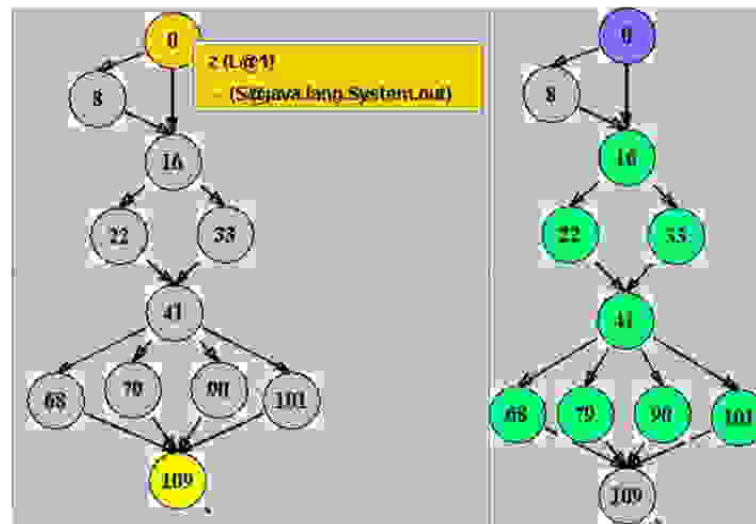


Figura 20: Visualização e Elementos Requeridos para todos-usos

No grafo da esquerda da Figura 20, é mostrada a lista das variáveis definidas no nó 0. Se o código Java for compilado com o parâmetro `-g`, o nome das variáveis é

mantido de forma a ser possível identificá-las (na lista, a variável z). Caso contrário, apresentar-se-á apenas o nome dado pelo compilador (no exemplo da variável Z , $L@1$).

O grafo da direita da Figura 20 mostra o nó de definição da variável escolhida na lista do grafo da esquerda (variável z), em azul, e seus usos.

3.2.2 Visualização dos Elementos Requeridos

A JaBUTi fornece uma lista com todos estes elementos, conforme o critério escolhido. Selecionando o item ‘Required Elements’, em ‘Visualization’, na barra de menu principal, é mostrada uma tela como a da Figura 21.

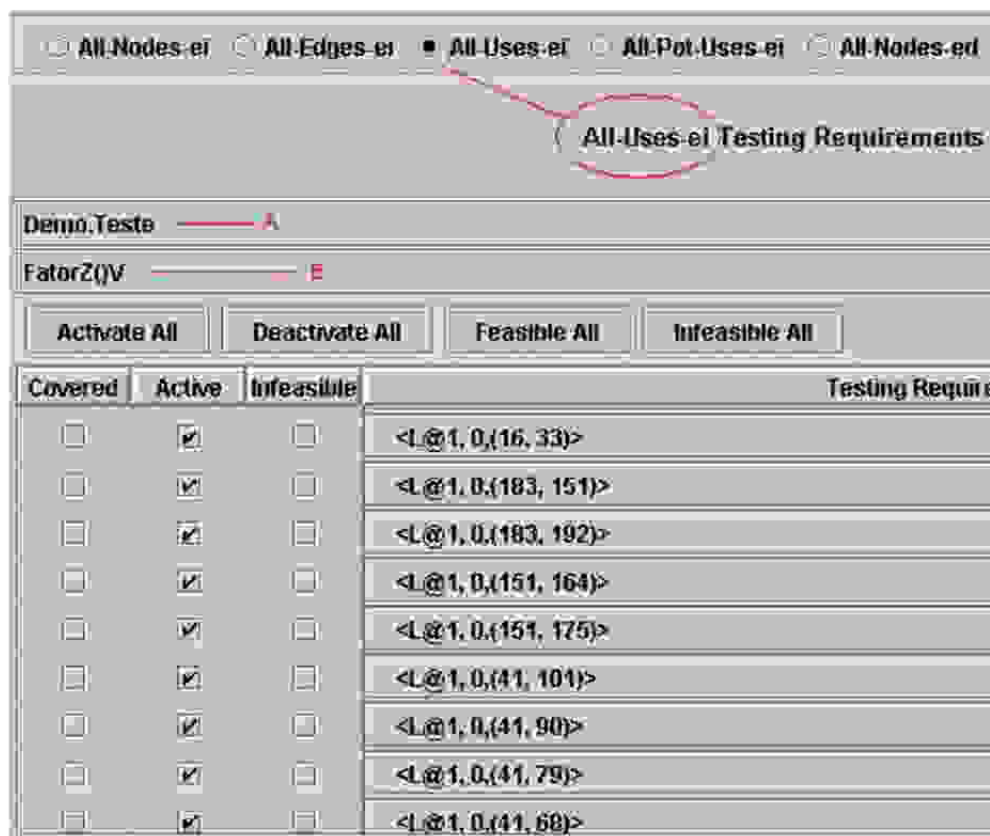


Figura 21: Lista de Elementos Requeridos

Circundado em vermelho está o critério ao qual pertencem os elementos requeridos da lista. No campo em que está a letra A , clicando-se com o mouse, surge uma lista com todas as classes previamente escolhidas para a análise na ferramenta. E no campo B , a lista dos métodos da classe escolhida.

A visualização desta lista pode ser importante para que se tenha uma clara visão sobre os elementos requeridos, o que nem sempre acontece analisando-se apenas o grafo.

Por exemplo, na Figura 20, pode não ficar claro se o uso da variável *z* é um nó (c-uso) ou arco (p-uso), ou ambos. No caso deste código, os elementos requeridos representam associações entre a definição no nó 0 e arcos, não havendo uso computacional. Este cenário fica evidente ao se observar a lista.

3.2.3 Análise do Código

Da mesma forma que os nós dos grafos são pintados de acordo com um peso, suas respectivas linhas de código bytecode também o são. E, apesar de não ser exigida a presença do código-fonte, caso ele esteja disponível e presente no mesmo diretório do bytecode, suas linhas também serão pintadas.

Com o auxílio do grafo, a análise sobre o código torna-se mais rápida. Na Figura 22 é apresentado este cenário. O grafo representa os fluxos de controle. Passado o mouse por um nó, sua respectiva linha de comando é mostrada, podendo-se identificá-la na tela do código. No exemplo, foi passado o mouse sobre o nó 95, para que se pudesse verificar qual sua respectiva linha (66), na tela da esquerda.

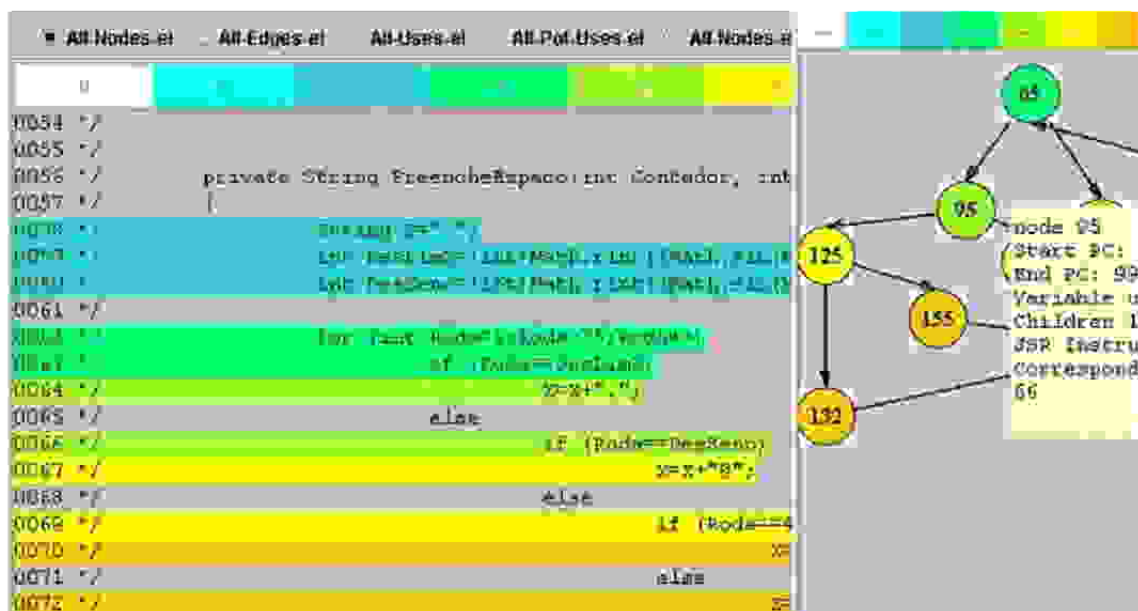


Figura 22: Análise do Código

O mesmo é aplicável ao bytecode, através do item *Start PC* e *End PC*, no quadro bege.

Assim como é possível verificar a lista de variáveis definidas no grafo, clicando-se com o botão direito do mouse sobre um nó colorido, o mesmo se aplica ao código.

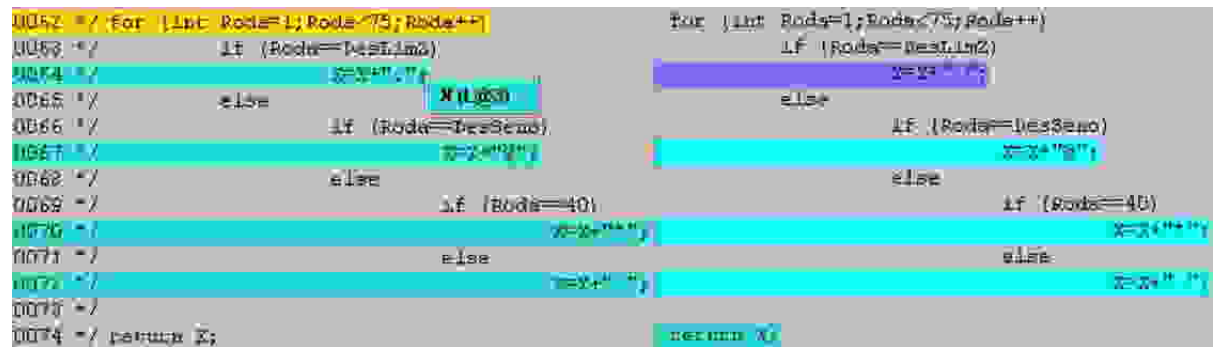


Figura 23: Lista de variáveis definidas vista pelo código

Na tela da esquerda da Figura 23, foi clicado o botão direito do mouse sobre a linha 64. Ao se escolher a única variável da lista, as linhas foram pintadas de forma a se mostrar em azul escuro a definição, e em claro os usos desta associação.

Outros dados relativos ao programa de teste podem ser analisados, escolhendo-se o item *Complexity Metrics* na opção *Visualization* da barra de menu. Dentre estes dados: número médio de parâmetros por método, tamanho médio dos métodos (código fonte e bytecode), profundidade da árvore de herança e outros.

3.3 Realização de Teste com a Ferramenta JaBUTi

Além de permitir um processo mais rápido da compreensão do código com a automatização da criação do grafo e as funcionalidades citadas, a ferramenta é provida da capacidade de detectar quais elementos requeridos foram exercitados durante a execução de um caso de teste. Ainda, gera dados estatísticos sobre a cobertura alcançada em relação a um método, a uma classe e ao total.

Para que a JaBUTi possa rastrear o caso de teste, em vez de se executar a classe principal, executa-se uma classe chamada *ProberLoader* e a classe principal é passada como seu parâmetro. A classe *ProberLoader* insere marcações no código a ser executado em cada bloco de instruções correspondente a um nó, e executa o programa. Através destas marcações consegue rastrear os caminhos exercitados. Tendo-se os caminhos exercitados e os elementos requeridos, a ferramenta determina quais destes elementos foram exercitados. Para executar um programa chamado Teste, se o comando para iniciá-lo fosse:

```
java Teste
```

O novo comando passaria a ser:

```
java -cp .;Jabuti-bin.zip probe.ProberLoader -P MeuTeste.jbt Teste
```

Este comando dado no prompt é dividido em quatro partes: a indicação do class-path com os pacotes necessários para que a ferramenta possa monitorar a execução do caso de teste, a chamada da classe responsável pelo monitoramento da execução (ProberLoader), o nome do projeto criado na JaBUTi que contém a classe estudada e o nome da classe que contém o método *main()* (Teste).

Com isso, a JaBUTi irá gerar um arquivo Mono.trc que conterá os dados colhidos para que não seja necessário executar o mesmo caso de teste a todo o momento em que se quiser realizar a análise sobre o resultado. Ao salvar o projeto, a partir dos dados do arquivo .trc, serão armazenadas no arquivo de projeto .jbt informações de cobertura dos critérios.

Após executado o teste, a opção “Update” da barra de menu principal tornar-se-á vermelha. Ao selecioná-la, os dados da ferramenta serão atualizados; isto é, o valor dos pesos serão alterados, cada elemento requerido exercitado será devidamente marcado (visível através da lista de elementos requeridos e pela cor branca no grafo e código).

Como exemplo, será focada atenção ao método *Exemplo()* do código do apêndice A. Serão realizados testes utilizando-se os critérios **todos-nós**, **todos-arcos** e **todos-usos**. Neste teste, supõe-se que foram determinadas as relações entre os parâmetros do método a ser testado e os dados de entrada de forma que as variáveis *x* e *z*, pedidas no início da execução (dados de entrada) correspondem às variáveis de mesmo nome no método. A variável *LimZ*, também pedida no início da execução, corresponde à variável *TextBox*.

Iniciando-se pelo critério **todos-nós**, o primeiro passo é escolher os casos de teste. Analisando o grafo (Figura 24), observa-se que não é possível exercitar todos os nós em apenas uma execução do programa, dada a presença de duas instruções *if-else*. Pelo menos duas fazer-se-ão necessárias. Ao se executar o primeiro caso de teste, pelo menos dois nós não serão exercitados: (60 ou 65) e (74 ou 81).

Deve-se, então, verificar quais as condições determinantes para que o fluxo con-virja para estes nós. Além destes, existe a presença de um *if* no primeiro nó, cuja condição deve ser analisada para que o fluxo passe pelo nó 17. O laço correspondente aos nós 91 e 99 sempre será exercitado, pois tendo-se em vista que a condição inicial é que a variável Contador seja menor que 361, observando o código, nota-se que tal variável sempre é

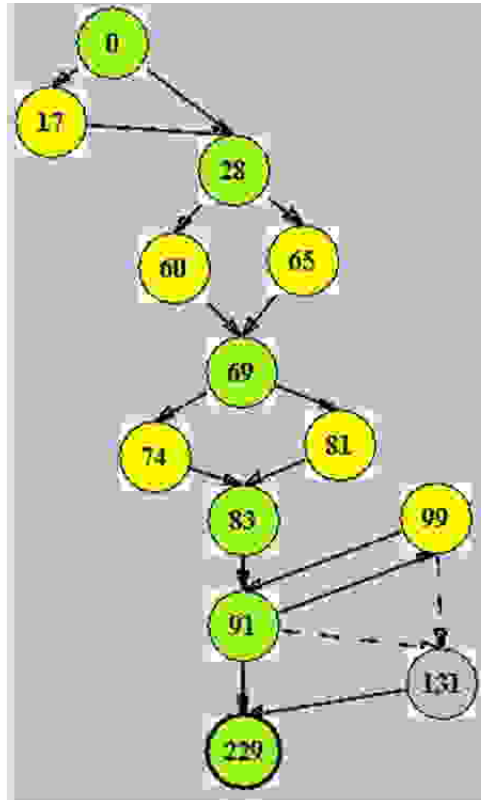


Figura 24: Análise Inicial do Método Exemplo()

iniciada com o valor 0.

Com esta análise, o caminho escolhido a ser exercitado pela primeira execução é: 0, 17, 28, 60, 69, 74, 83, 91, 99, 91, 229 (não serão exercitados os nós 65 e 81). Para tanto, a variável *TextBox* tem que ser maior que 360 (condição no nó 0), a variável *x* tem que ser diferente de 0 (condição no nó 28) e a variável *z* deve ser maior que 1 (condição no nó 69). Foram escolhidas as seguintes entradas:

- $x = 5$, $z = 10$, *TextBox* = 380. Caminho: 0,17,28,60,69,74,83,91,99,91,229.

Na Figura 25 é mostrado o grafo atualizado, após a execução deste caso de teste, deflagrando que os nós 65 e 81 não foram exercitados, como o esperado.

É importante ressaltar que o caso de teste não é formado apenas pelos dados de entrada e, conseqüentemente, a simples execução do programa em si não possui significado. O caso de teste é formado também pela saída esperada, e após a execução, deve-se verificar se a saída obtida é a saída esperada, constituindo o verdadeiro significado do teste. No exemplo, a saída é um grafo senóide apresentando um tracejado correspondente à posição do seno da variável *TextBox*. Como na especificação, valores maiores que 360

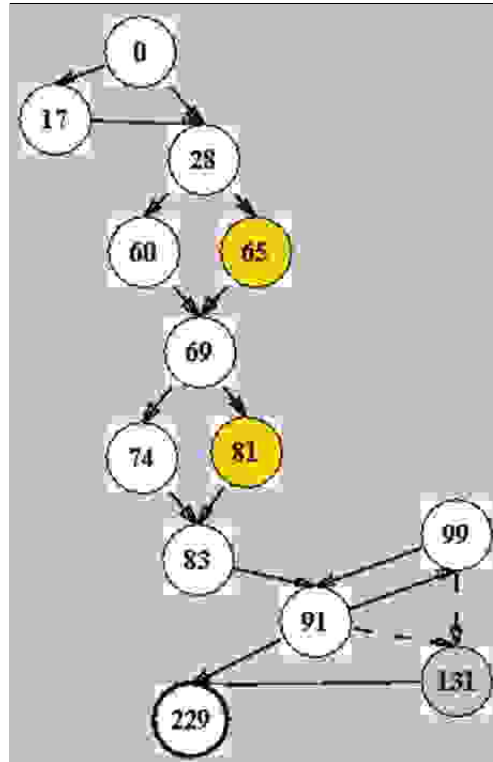


Figura 25: Critério Todos-Nós:Resultado do Primeiro Caso de Teste

são considerados 360 para esta variável, e sabendo-se que o seno de 360 é 0, a saída esperada e a saída obtida são coincidentes.

Um próximo caso de teste irá exercitar os nós restantes. Para isto, foram escolhidas as seguintes entradas:

- $x = 0$, $z = 1$, $TestBor = 380$. Caminho: 0,17,28,65,69,81,83,91,99,91,229.

Após sua execução, verificar-se-á que todos os nós estão brancos. A saída obtida correspondeu à saída esperada.

Em seguida aos testes realizados com o critério todos-nós, clicando-se na opção **todos-arcos**, o grafo aponta um nó de decisão não exercitado (grafo à esquerda na Figura 26), significando que os testes anteriormente realizados cobriram a maior parte dos elementos requeridos para o critério todos-arcos. Ao clicar sobre o nó de cor amarela, será apresentado o arco não exercitado (0,28), como mostra o grafo da direita da Figura 26.

Para exercitá-lo, faz-se necessário que o valor do dado de entrada correspondente a LimZ seja menor ou igual a 360. Um possível caso de teste pode ser:

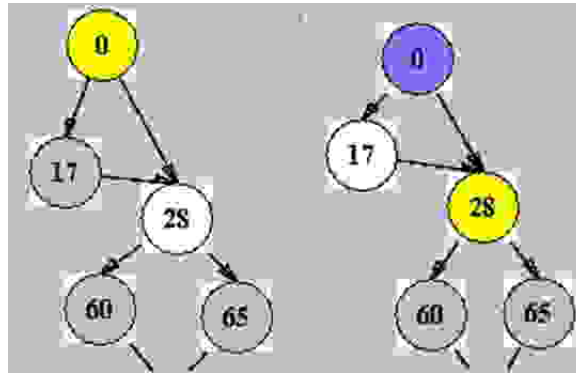


Figura 26: Critério Todos-Arcos:Resultado do Primeiro Caso de Teste

- $x = 5, z = 10, TextBox = 90$. Caminho: 0,28,60,69,74,83,91,99,91,229.

Ao se observar a saída obtida (gráfico à esquerda na Figura 27), observa-se que o resultado não condiz com o esperado. Sendo o seno de 90 igual a 1, espera-se que o resultado seja igual ao da direita da mesma Figura.

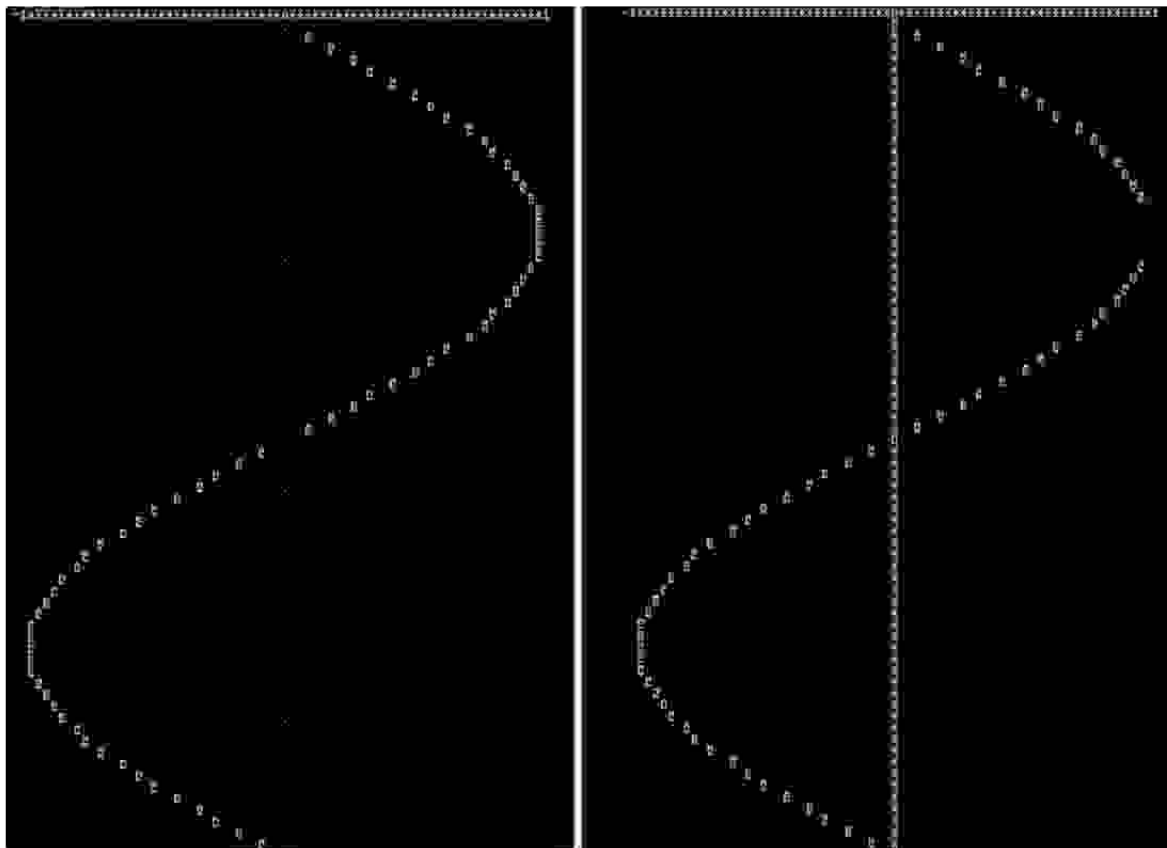


Figura 27: Saída Obtida e Esperada

Observa-se que nem todos os elementos do subdomínio representado por todas as entradas que exercitam o arco (0,28) são capazes de identificar o defeito mostrado.

Os senos de 180, 360, 0 e múltiplos negativos de 180 são iguais a 0, de maneira que dados de entrada com estes valores para *LimZ* não detectam o defeito. Portanto, apesar do exemplo demonstrar que se o critério todos-nós for satisfeito não necessariamente o critério todos-arcos será satisfeito, *este exemplo não prova que o critério todos-arcos encontre mais defeitos que o critério todos-nós*.

Terminados os testes com base no critério todos-arcos, ao se selecionar o critério todos-usos na Ferramenta JaBUTi, observar-se-á quatro elementos requeridos não exercitados: $\langle L@4, 0.(91.229) \rangle$, $\langle L@2, 65(69.74) \rangle$, $\langle L@1, 0.74 \rangle$ e $\langle L@2, 65.74 \rangle$.

- $L@1$ corresponde à variável *x*;
- $L@2$ corresponde à variável *z*;
- $L@4$ corresponde à variável *Contador*.

O primeiro elemento requerido, relacionado à variável *Contador*, é infactível. Seu uso é o arco de saída do laço *while*. Porém, como o laço é exercitado obrigatoriamente e dentro dele há uma redefinição de *Contador*, é impossível exercitar a referida associação.

Para que o elemento requerido relacionado à variável *x* seja exercitado, deve-se escolher um caminho que não passe pelo nó 25, onde há sua redefinição, e passe pelo nó 74.

E para que os elementos requeridos relacionados à variável *z* sejam exercitados, para a associação entre o nó 0 e 74, deve-se escolher um caminho que não passe pelo nó 27, onde há sua redefinição, e passe pelo nó 74; e um caminho que passe pelos nós 65 e 74. Observa-se que para que os dois elementos requeridos restantes relacionados à variável *z*, serão necessárias duas execuções.

Dois possíveis casos de teste que cubram todos os elementos requeridos são:

- $x = 5, z = 0, TextBox = 270$. Caminho: 0,28,60,69,81,83,91,99,91,229;
- $x = 0, z = 5, TextBox = 180$. Caminho: 0,28,65,69,74,83,91,99,91,229.

Porém, durante a execução do segundo caso de teste, um novo defeito será descoberto. Exercitando-se o nó 74, sendo $x = 0$, ocorrerá uma divisão por zero.

Ao se executar um ou mais casos de teste, a JaBUTi calcula a percentagem de elementos requeridos cobertos até o momento. Estes dados estatísticos podem ser visualizados no item *Required Elements* na opção *Visualization* do menu principal ou na opção *Summary* em que pode-se escolher verificar os dados por critério, classe ou método.

Apesar da JaBUTi armazenar todos os casos de teste, é possível excluir ou desativar quaisquer casos de teste que se queira, no item *Report by Test Case* na opção *Test Case* da barra principal.

Por fim, uma outra função fornecida pela ferramenta é o **slicing tool**. Ao se escolher este item na opção *Tools* do menu principal, será pedido ao usuário que selecione até dois casos de teste que falharam em detectar o defeito e até dois bem sucedidos em encontrar o mesmo defeito. Com isto, serão apresentados os locais mais prováveis onde o defeito pode estar presente. O seu mecanismo funciona da seguinte maneira: é sobreposto o caminho realizado pelo caso de teste bem sucedido com o que falhou em detectar um defeito. Os nós pertencentes ao caso de teste que foi bem sucedido em detectar um defeito e que não estão presentes no caminho do caso de teste que falhou em detectar o defeito são os mais prováveis locais onde se encontra o defeito.

Assim, escolhida a opção *Slicing Tool*, o usuário deve entrar na tela do item *Report by Test Case* da opção *Test Case*. No exemplo mostrado, o terceiro e quarto caso de teste detectaram um defeito ao não aparecer o traço na posição correspondente ao seno de 90, isto é, 1. Já, no primeiro e segundo casos de teste este defeito não foi observado. Conforme a Figura 28, pode-se escolher o caso de teste 1 e 2 como os que falharam em detectar o defeito e o caso de teste 3 e 4 como os que foram bem sucedidos em detectar o defeito. Após escolhidos, clica-se na opção *Update*.

All Nodes of Coverage per Test Case				
Success	Fail	Test Case	Host	Total Coverage
☐	☒	0001	localhost	22 of 45
☐	☒	0002	localhost	22 of 45
☒	☐	0003	localhost	21 of 45
☒	☐	0004	localhost	21 of 45
☐	☐	0005	localhost	7 of 45

Figura 28: Slicing Tool: Escolha de Elementos Requeridos

Ao retornar à tela do código-fonte, e escolhida a opção *Update*, estarão marcados os locais mais prováveis de onde se encontra o defeito, como mostrado na Figura 29.

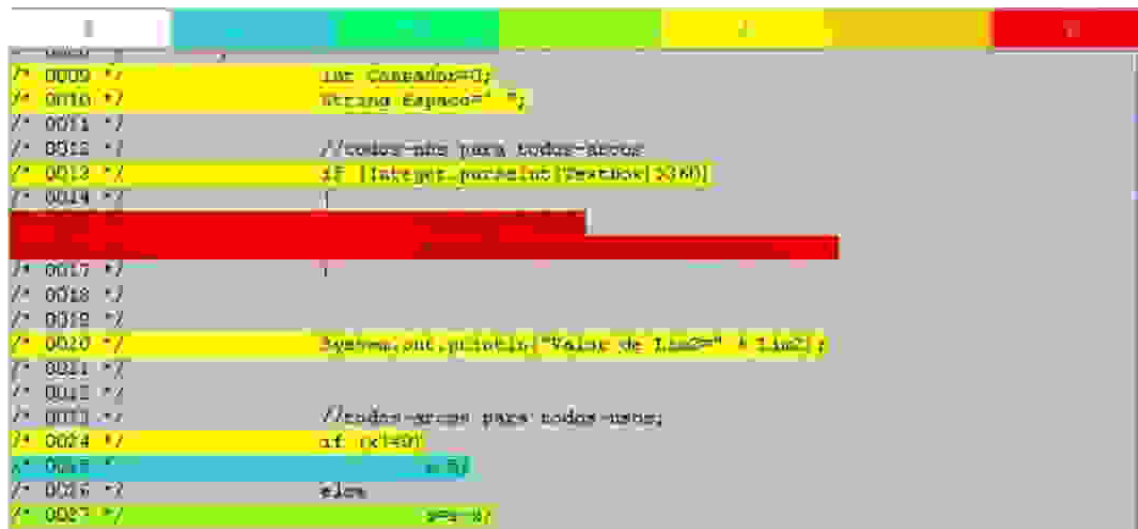


Figura 29: Slicing Tool: Local Provável do Defeito

Conforme o peso distribuído, nota-se que o defeito tem maior probabilidade de estar entre as linhas 15 e 16. E de fato, a linha 16 deveria estar posicionada após o bloco do if, na linha 18, sendo este o defeito do programa que leva a se marcar a posição errada para o seno de qualquer valor não múltiplo de 180, não maior que 360 e diferente de 0.

3.4 Conclusão

Neste capítulo, foram abordados conceitos que permitiram o entendimento de como a JaBUTi prioriza os elementos requeridos, através da distribuição de pesos que permitem a usuário visualizar quais são os elementos que, uma vez exercitados, induzem ao exercício de um maior número de outros elementos. Foram apresentados os principais processos automatizados e funcionalidades da Ferramenta JaBUTi, e através de um exemplo prático, como analisar um código através de sua interface gráfica.

4 *Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional*

Este capítulo apresenta os principais conceitos relacionados aos critérios de ABDR de unidade criados por Spoto (2000), abordando a adaptação da representação dos grafos, um exemplo de utilização dos critérios e a análise de inclusão.

4.1 Conceitos Básicos

Dada a natureza persistente dos dados manipulados pelas aplicações de banco de dados relacional (ABDR), a classificação de testes de unidade, integração e sistema foram adaptados em Spoto (2000) para realizar testes estruturais entre programas, pois existem cenários em que dois ou mais programas utilizam a mesma base de dados, concorrentemente.

Desta forma, denomina-se módulo um programa de aplicação de banco de dados relacional, e unidade de programa a menor parte executável de um programa (métodos, procedimentos e funções). Com isto, classificam-se os testes em intra-modulares e inter-modulares. O primeiro caracteriza os testes de unidade e integração entre as unidades de um mesmo módulo, e os testes inter-modulares são aqueles utilizados para os testes considerando-se o fluxo de dados entre os módulos de programa (integração entre módulos).

Para as ABDR, foram necessárias, também, adaptações na representação dos grafos, visto que os critérios abrangem o uso dos comandos SQL. Assim, adota-se que para cada comando SQL de manipulação de dados (*insert*, *delete*, *update* e *select*), incluindo-se *commit* e *rollback*, é gerado um nó diferenciado, representado por um quadrado no

diagrama do grafo (SPOTO, 2000).

Para o trecho de código da Figura 30, o grafo da Figura 31 é gerado.

Os nós 6, 10, 14, 22 e 29 representam comandos SQL de linguagem de manipulação de dados e não se inclui instruções da aplicação.

Outros conceitos importantes para adaptar os testes estruturais para ABDR são descritos em Spoto (2000). Dentre eles, a classificação das variáveis em:

- Variáveis de programa são aquelas normalmente utilizadas na programação e que não se relacionam diretamente com os comandos SQL;
- Variáveis **host** realizam o intercâmbio de valores entre o programa e a base de dados e podem ser usadas tanto no programa como nos comandos SQL. No exemplo, a variável *Nome* armazena um valor da base de dados, caracterizando-o como *host*;
- Variáveis **tabela** são as próprias tabelas do banco de dados, manipuladas pela aplicação. No exemplo, a variável tabela é *Cliente*.

As variáveis *host* e *tabela* também sofrem os dois tipos de ação: definição e uso.

Para as variáveis **host**, além das mesmas definições de uma variável de programa, como mostrado no nó 1, também são definidas quando estiverem dentro de uma cláusula INTO do comando SELECT ou FETCH, por exemplo:

O seu uso ocorre na condição da cláusula WHERE dos comandos SELECT, DELETE e UPDATE, do lado direito do comando de atribuição da cláusula SET do comando UPDATE e na cláusula VALUE do comando INSERT.

Para as variáveis **tabela**, ocorre uma definição persistente quando um valor é armazenado em memória secundária, modificando o estado destas variáveis. Assim, há definição quando a variável tabela estiver presente em uma cláusula INTO do comando INSERT, em uma cláusula FROM do comando DELETE, do lado esquerdo da cláusula SET do comando UPDATE e, no caso de uma visão, quando estiver no lado direito da cláusula CREATE VIEW. E ocorrerá um t-uso quando a variável estiver na cláusula FROM dos comandos SELECT e DELETE, em uma cláusula INTO do comando INSERT, do lado

esquerdo da cláusula SET do comando UPDATE, na cláusula WHERE dos comandos executáveis DELETE, UPDATE e SELECT (SPOTO, 2000).

Como ocorre definição de variável tabela na execução dos comandos *insert*, *delete* e *update*, e ocorre t-uso na execução dos comandos *select*, *insert*, *delete* e *update*, observa-se que em um mesmo comando SQL haverá um t-uso e uma definição quando executados os comandos *insert*, *delete* e *update*. Por exemplo:

```
insert into cliente values (1, 'Murillo')
```

Primeiro ocorre o seu uso, pois a tabela (ou as páginas de dados onde será inserido o novo registro) é trazida para o buffer, para em seguida ser inserido o valor no banco de dados, ocorrendo a definição. Deve-se observar que a persistência da definição apenas ocorre após o COMMIT, isto é, quando o valor é armazenado em memória secundária. Enquanto este último comando não for executado, diz-se que a ação é uma definição por referência.

Para o t-uso, deve-se considerar o arco formado pelo nó de comando SQL (*select*, *insert*, *update* e *delete*) e o nó, ou nós, de saída.

Outro importante conceito é o de dtu-caminho, que é um du-caminho com relação a variável tabela no enfoque de variável persistente (SPOTO, 2000).

4.2 Critérios de Teste

Os critérios a serem apresentados representam o escopo de teste de unidade dos testes intra-modulares. Seja Π um conjunto de caminhos completos (do nó de entrada ao de saída) de um grafo de programa e Γ um conjunto de tuplas $\{\tau_1, \tau_2, \dots, \tau_c\}$ pertencentes a uma dada tabela t (SPOTO, 2000):

- Critério **todos-t-usos** (a ser implementado na Ferramenta JaBUTi): requer para todas as associações definição-t-uso $\langle \langle l, i \rangle, (j, k), t \rangle$ que pelo menos um caminho $\pi = (n_{in}, \dots, l, \dots, i, \dots, j, k, \dots, n_{out}) \in \Pi$, livre de definição persistente de $\langle l, i \rangle$ até (j, k) c.r.a. t , seja exercitado pelo menos uma vez, para a mesma tupla $\tau \in \Gamma$;
- Critérios **todos-dtu-caminhos**: requer que todos os dtu-caminhos $(n_l, \dots, n_i, \dots, n_j, n_k)$ sejam executados pelo menos uma vez. A associação deve ser exercitada para a mesma tupla $\tau \in \Gamma$.

Em todos-t-usos, j representa o nó com a presença de um dos quatro comando SQL que caracterizam o t-uso e k o nó sucessor ou nós sucessores. Outros critérios para utilização em testes de integração e inter-modulares são encontrados em Spoto (2000). O nó l representa a definição e i , o nó de *commit*.

4.3 Casos de Teste

Durante a elaboração e execução dos casos de teste, deve-se ter em mente que o objetivo não é puramente testar se as instruções funcionam, mas se o comportamento do programa está de acordo com as ações realizadas no banco de dados. É possível, também, verificar as dependências funcionais e inconsistências na base de dados. Isto implica que não basta se ater à análise do código, pois a análise dos casos de teste vão além, observando-se o comportamento da própria base de dados, como citado por Chays et al. (2000): “... ao checar os resultados, o testador deve prestar atenção ao estado da base de dados após executar a aplicação”.

4.4 Exemplo Prático

Para ilustrar a aplicação dos critérios de ABDR para teste de unidade, foi utilizada uma base de dados que possui uma tabela (cliente) previamente populacionada com dez registros (Figura 32), cujos atributos são: codcliente (chave primária com auto-incrementação), nome e CPF (unicidade com permissão para campo nulo).

Serão mostrados alguns dos defeitos que podem ser detectados por estes critérios, alterando-se as restrições impostas na tabela cliente, como poderá ser verificado na seção *Criação e Análise dos Casos de Teste*. É importante citar que o uso dos critérios todos-t-usos e todos-dtu-caminhos, neste capítulo, objetiva mostrar a dinâmica de suas aplicações, para melhor compreensão de como implementar um critério de teste em aplicação de bancos de dados.

O código testado é apresentado na Figura 33.

O grafo da Figura 34 representa o código exemplo da Figura 33.

O nó 20 (Figura 34) representa um comando *insert*. No nó 24, há um comando

delete. O nó 26 representa o comando *commit*. Já, em 29 e 32, executa-se uma *query* em cada nó. As arestas pontilhadas representam fluxos dependentes de exceções.

O tópico a seguir apresenta uma abordagem de como se criar casos de teste para aplicações de banco de dados relacional.

4.4.1 Análise da Aplicação e Base de Dados

Para que se possa gerar os casos de teste em que se utilizará o critério todos-t-usos, determina-se inicialmente os pares definição-t-uso. Existem dois pares de nós em que é realizada a definição persistente, conforme a Tabela 3. Os comandos SQL que caracterizam o t-uso estão presentes em quatro pares de nós, formando, assim, oito arcos, dado que para cada um dos nós, há dois arcos de saída:

Tabela 3: definições e t-usos do código da Figura 33

Definição	t-uso			
$\langle 20, 26 \rangle$	(20, 24)	(24, 25)	(29, 30)	(32, 33)
$\langle 24, 26 \rangle$	(20, 40)	(24, 40)	(29, 40)	(32, 40)

Através da análise do grafo, observa-se que existem pares definição-uso das instruções INSERT e DELETE para os comandos SELECT, mas por estarem todas dentro de um laço while, existirão definições globais persistentes das duas primeiras instruções com uso nelas próprias na próxima execução do laço. Por exemplo, utilizando-se o INSERT da linha 20 (nó 19 da Figura 34), tem-se primeiro um uso, como explicado anteriormente em *Conceitos Básicos*, página 57, em seguida a sua definição. Exercitados os demais blocos, no momento em que o fluxo convergir novamente para o nó 14, supondo-se que a condição *while* mantenha-se verdadeira, durante a execução do nó 20 ou 24, haverá um novo uso, constituindo-se o par definição-t-uso.

Existem, assim, dezesseis pares definição-t-uso, sendo primários aqueles que fazem parte do fluxo esperado de controle e secundários os que geram desvio do fluxo para o tratamento de exceções, conforme a Tabela 4.

O par em vermelho é infactível, pois existe restrição de unicidade para o atributo CPF e como se deve exercer a mesma tupla, sempre ocorrerá uma exceção ao se realizar duas inserções com estes mesmos valores. Porém, há de se notar que nestas situações, ainda que verificada a restrição de unicidade, em um segundo momento, deve-se realizar

Tabela 4: pares definição-t-uso

Pares definição-t-uso			
Primários		Secundários	
[< 20, 26 >, (29, 30)]	[< 24, 26 >, (29, 30)]	[< 20, 26 >, (29, 40)]	[< 24, 26 >, (29, 40)]
[< 20, 26 >, (32, 33)]	[< 24, 26 >, (32, 33)]	[< 20, 26 >, (32, 40)]	[< 24, 26 >, (32, 40)]
[< 20, 26 >, (20, 21)]	[< 24, 26 >, (20, 21)]	[< 20, 26 >, (20, 40)]	[< 24, 26 >, (20, 40)]
[< 20, 26 >, (24, 25)]	[< 24, 26 >, (24, 25)]	[< 20, 26 >, (24, 40)]	[< 24, 26 >, (24, 40)]

o mesmo teste com tuplas diferentes para que se observe se o comportamento do banco, além da restrição de unicidade, é o esperado durante a execução do comando SQL.

Para o critério todos-dtu-caminhos, existem 36 caminhos a serem levados em consideração durante a criação dos casos de teste (Tabela 5).

Os três primeiros caminhos primários em vermelho, pelo mesmo motivo que o apresentado para o quadro de pares definição-t-uso, são ineficazes. Os demais são ineficazes porque o nó 36 define o fim do laço não sendo possível exercer os nós seguintes a partir do 15.

Ao serem pensados os casos de teste, deve-se ter em mente as regras aplicadas às tabelas utilizadas para que se possa definir o que será observado após a execução de um caso de teste. Na simulação realizada, a base de dados possui uma tabela cliente cujas *constraints* são: uma chave primária em codcliente e unicidade em CPF. Apenas neste último, é permitido campo nulo.

Assim, além da análise sobre a corretude da execução do aplicativo, deve ser verificado:

- Nos casos em que tanto a definição como o uso correspondem a *insert*, se foi inserido incorretamente os mesmos atributos únicos (codcliente e CPF);
- Se o comando *insert* permite valores nulos para CPF;
- Se o comando *insert* permite valores nulos para codcliente;

Para o primeiro item de verificação, o esperado é que ocorra uma exceção ao se realizar duas inserções (contanto que não haja uma redefinição entre estas ações). É por isto que não será derrubado o banco de dados para se testar esta situação com presença de fluxo secundário, já que o resultado ficaria mascarado.

Tabela 5: dtu-caminhos do código da Figura 33

dtu-caminhos	
Primários	Secundários
(20, 21, 26, 27, 29, 30, 37, 14, 15, 18, 20, 21)	(20, 21, 26, 27, 29, 30, 37, 14, 15, 18, 20, 40)
(20, 21, 26, 27, 35, 37, 14, 15, 18, 20, 21)	(20, 21, 26, 27, 35, 37, 14, 15, 18, 20, 40)
(20, 21, 26, 27, 32, 33, 37, 14, 15, 18, 20, 21)	(20, 21, 26, 27, 32, 33, 37, 14, 15, 18, 20, 40)
(20, 21, 26, 27, 36, 37, 14, 15, 18, 20, 21)	(20, 21, 26, 27, 36, 37, 14, 15, 18, 20, 40)
(24, 25, 26, 27, 29, 30, 37, 14, 15, 23, 24, 25)	(24, 25, 26, 27, 29, 30, 37, 14, 15, 23, 24, 40)
(24, 25, 26, 27, 35, 37, 14, 15, 23, 24, 25)	(24, 25, 26, 27, 35, 37, 14, 15, 23, 24, 40)
(24, 25, 26, 27, 32, 33, 37, 14, 15, 23, 24, 25)	(24, 25, 26, 27, 32, 33, 37, 14, 15, 23, 24, 40)
(24, 25, 26, 27, 36, 37, 14, 15, 23, 24, 25)	(24, 25, 26, 27, 36, 37, 14, 15, 23, 24, 40)
(20, 21, 26, 27, 29, 30, 37, 14, 15, 23, 24, 25)	(20, 21, 26, 27, 29, 30, 37, 14, 15, 23, 24, 40)
(20, 21, 26, 27, 35, 37, 14, 15, 23, 24, 25)	(20, 21, 26, 27, 35, 37, 14, 15, 23, 24, 40)
(20, 21, 26, 27, 32, 33, 37, 14, 15, 23, 24, 25)	(20, 21, 26, 27, 32, 33, 37, 14, 15, 23, 24, 40)
(20, 21, 26, 27, 36, 37, 14, 15, 23, 24, 25)	(20, 21, 26, 27, 36, 37, 14, 15, 23, 24, 40)
(24, 25, 26, 27, 29, 30, 37, 14, 15, 18, 20, 21)	(24, 25, 26, 27, 29, 30, 37, 14, 15, 18, 20, 40)
(24, 25, 26, 27, 35, 37, 14, 15, 18, 20, 21)	(24, 25, 26, 27, 35, 37, 14, 15, 18, 20, 40)
(24, 25, 26, 27, 32, 33, 37, 14, 15, 18, 20, 21)	(24, 25, 26, 27, 32, 33, 37, 14, 15, 18, 20, 40)
(24, 25, 26, 27, 36, 37, 14, 15, 18, 20, 21)	(24, 25, 26, 27, 36, 37, 14, 15, 18, 20, 40)
	(20, 21, 26, 27, 29, 40)
	(20, 21, 26, 27, 32, 40)
	(24, 25, 26, 27, 29, 40)
	(24, 25, 26, 27, 32, 40)

4.4.2 Criação e Análise dos Casos de Teste

Com base na análise sobre o aplicativo e a base de dados, é possível gerar os casos de teste. Como as funcionalidades estão todas dentro de um laço, para os pares definição-t-uso primários, todos os pares factíveis podem ser exercitados em apenas um caso de teste, como no cenário a seguir:

- Inserir, consulta ordenada por nome, apagar, consulta ordenada por nome, apagar, consulta ordenada por CPF, inserir, consulta ordenada por CPF, apagar, sair;

Como os tratamentos de exceção terminam o programa, conseqüentemente haverá um caso de teste para cada par secundário, como demonstrado a seguir:

- Inserir e consulta ordenada por nome;
- Inserir e consulta ordenada por CPF;
- Inserir, voltar e inserir;

- Inserir, voltar e apagar;
- Apagar e consulta ordenada por nome;
- Apagar e consulta ordenada por CPF;
- Apagar, voltar e inserir;
- Apagar, voltar e apagar.

O ambiente inicialmente elaborado não possui defeitos dentro das restrições previstas para o banco de dados, mas as mensagens de sucesso ou falha da inserção e eliminação não foram bem projetadas de maneira que, logo em seguida à mensagem, a tela é apagada não sendo possível definir se a ação foi concluída com êxito. Se uma inserção, seguida de outra inserção ocorrer, a única forma de averiguar se a base de dados possui duas tuplas idênticas (o que não pode acontecer) é observando o banco de dados.

Outros defeitos poderiam ser detectados com este teste observando-se a base de dados, como um caso em que a regra de unicidade no campo CPF ou a chave primária, mesmo que requeridas, não fossem implementadas.

Nota-se que mesmo não ocorrendo um problema no aplicativo, a base de dados poderia estar inconsistente, evidenciando-se a importância de se avaliar o próprio banco de dados.

Com os critérios todos-t-usos, 8 dtu-caminhos foram exercitados, sendo quatro primários e quatro secundários, como mostra a Tabela 6.

Tabela 6: dtu-caminhos exercitados com o critério todos-t-usos

Primários	Secundários
(24, 25, 26, 27, 29, 30, 37, 14, 15, 23, 24, 25)	(20, 21, 26, 27, 29, 40)
(20, 21, 26, 27, 29, 30, 37, 14, 15, 23, 24, 25)	(20, 21, 26, 27, 32, 40)
(20, 21, 26, 27, 32, 33, 37, 14, 15, 23, 24, 25)	(24, 25, 26, 27, 29, 40)
(24, 25, 26, 27, 32, 33, 37, 14, 15, 18, 20, 21)	(24, 25, 26, 27, 32, 40)

Para exercitar todos os requisitos de todos-dtu-caminhos primários, serão separados, para melhor visualização, os conjuntos de funções que devem ser executadas, apesar de poderem ser feitos em apenas um caso de teste:

- Apagar, voltar e apagar;

- Apagar, consulta ordenada por CPF e apagar;
- Inserir, voltar e apagar;
- Apagar, consulta ordenada por nome e inserir;
- Apagar, voltar e inserir.

Os du-caminhos secundários são exercitáveis para as mesmas ações anteriores com o banco de dados derrubado no meio do processo, e com o seguinte conjunto de ações:

- Inserir, voltar e inserir;
- Inserir, consulta ordenada por nome e inserir;
- Inserir, consulta ordenada por CPF e inserir;
- Apagar, consulta ordenada por nome e apagar;
- Inserir, consulta ordenada por nome e apagar;
- Inserir, consulta ordenada por CPF e apagar;
- Apagar, consulta ordenada por CPF e inserir.

Quando as regras são administradas pelo Sistema Gerenciador de Banco de Dados os testes podem detectar erros como, por exemplo, a hipótese de se esquecer de implementar a unicidade no campo CPF. Porém, em sistemas em que a validação de um atributo é feita pela aplicação, os testes estruturais tornam-se ainda mais críticos, pelo fato de que entre uma definição e uma ação irão existir nós que deverão ser exercitados para cumprir as verificações das regras. Isso aumenta a complexidade da unidade a ser testada, ou conjunto de unidades caso os processos de validação façam parte de funções separadas, que influenciarão diretamente sobre a integridade do banco de dados e as dependências funcionais.

4.5 Análise de Inclusão

Spoto (2000) analisa a relação entre os critérios todos-t-usos e todos-dtu-caminhos, e entre estes e a Família de Critérios Potenciais Usos. Sob o escopo dos testes de unidade, são particularmente importantes os dois primeiros teoremas e suas respectivas provas.

O primeiro teorema cita que os critérios FCPU são incomparáveis aos critérios todos-t-usos e todos-dtu-caminhos, sendo demonstrado que os critérios de Spoto exercitam variáveis persistentes não necessariamente contempladas pelos critérios tradicionais. É também demonstrado que os critérios todos-potenciais-usos não incluem os critérios todos-t-usos sendo que este último requer o uso da mesma tupla na definição e no t-uso e o primeiro não. Por fim, é demonstrado que o critério todos-t-usos não inclui o critério todos-potenciais-usos porque existem partes do programa que não envolvem nós SQL.

O segundo teorema cita que o critério todos-dtu-caminhos inclui parcialmente o critério todos-t-usos. Na presença de dtu-caminhos não executáveis, os critérios são incomparáveis, já em uma situação em que todos os dtu-caminhos são executáveis, o critério todos-dtu-caminhos inclui o todos-t-usos. Se existe apenas um caminho entre uma definição persistente e um t-uso, ao se satisfazer o critério todos-dtu-caminhos, o critério todos-t-usos também será satisfeito. Quando houver mais de um caminho, o critério todos-dtu-caminhos exercitará os caminhos que seriam exercidos pelo critério todos-t-usos e outros que não seriam. Com isto, diz-se que o critério todos-dtu-caminhos inclui o critério todos-t-usos, mas o critério todos-t-usos não inclui o critério todos-dtu-caminhos. Nas Figuras 35 e 36 é apresentada a relação de inclusão entre os critérios de ABDR.

4.6 Conclusão

Este capítulo apresentou os principais conceitos relacionados a aplicações de bancos de dados relacionais, abordando as definições de variáveis de aplicação, variáveis host e tabela, t-uso e dtu-caminho, as adaptações na representação de grafos para ABDR e adaptações do conceito de teste de unidade, integração e sistema.

Em seguida, foram apresentados os critérios de Spoto (2000) e um exemplo de suas aplicabilidades utilizando-se uma aplicação real com casos de teste que permitiram a visualização de alguns dos defeitos que podem ser detectados.

A análise de inclusão mostrou a relação entre os critérios de ABDR para testes de unidade e um cenário em que o critério todos-dtu-caminhos não inclui o todos-t-usos, para dtu-caminhos não executáveis.

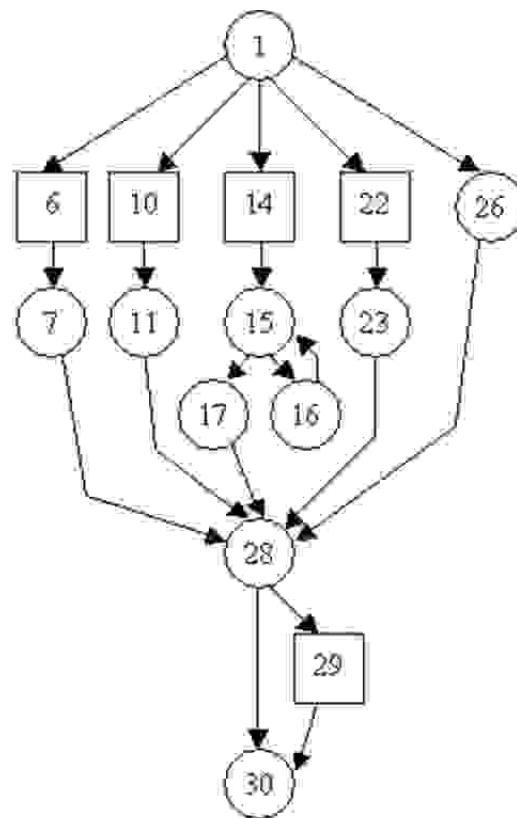


Figura 31: grafo para aplicação ABDR.

	codcliente	nome	CPF
1		1 Ana	111.111.111-11
2		2 Maria	222.222.222-22
3		3 Fabiana	333.333.333-33
4		4 Márcio	444.444.444-44
5		5 Cláudio	555.555.555-55
6		6 Pietra	666.666.666-66
7		7 Pedro	777.777.777-77
8		8 Joana	888.888.888-88
9		9 Denise	999.999.999-99
10		10 Renaldo	777.000.117-11

Figura 32: população da tabela cliente

```

public void Funcao() throws Exception
{
1  String URL="jdbc:odbc:mySql";
2  String Nome="Erro";
3  String CPF="Erro";
4  String Consulta1="select * from cliente order by nome";
5  String Consulta2="select * from codcliente order by CPF";
6  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
7  Connection Con= DriverManager.getConnection(URL,"","");
8  Statement stmt=Con.createStatement();
9  ResultSet rs;
10 int Contador=1, CodCliente=0;
11 int Opcao;
12 boolean Flag=true;
13 try
14 {
15     Con.setAutoCommit(false);
16     while (Flag)
17     {
18         Opcao=TelaOp();
19         CodCliente=Integer.parseInt(TelaNome("Codigo"));
20         switch (Opcao)
21         {
22             case 0:
23                 Nome=TelaNome("nome"), CPF=TelaNome("CPF");
24                 String Insert = "insert into cliente values(" +
25                     CodCliente + "," + Nome + "," + CPF + ")";
26                 stmt.execute(Insert);
27                 System.out.println("Operação insert realizada");
28                 break;
29             case 1:
30                 String Apaga="delete from cliente where nome='"+
31                     Nome + "'";
32                 stmt.execute(Apaga);
33                 System.out.println("Operação delete realizada");
34             }
35         Con.commit();
36         Opcao=TelaConsulta();
37     }
38     switch (Opcao)
39     {
40         case 0:
41             rs=stmt.executeQuery(Consulta1);
42             ApresentaRes(rs);
43             break;
44         case 1:
45             rs=stmt.executeQuery(Consulta2);
46             ApresentaRes(rs);
47             break;
48         case 2:
49             break;
50         default:
51             Flag=false;
52     }
53     Contador=Contador+1;
54 }
55 stmt.close();
56 Con.close();
57 }
58 catch (SQLException e)
59 {
60     System.out.println(e.getSQLState());
61     System.out.println(e.getMessage());
62     System.out.println(e.getErrorCode());
63     System.out.println();
64     e.getNextException();
65     stmt.close();
66     Con.close();
67     Funcao();
68 }
69 catch (Exception e)
70 {
71     System.out.println("Ocorreu algum erro");
72     System.out.println(e);
73     stmt.close();
74     Con.close();
75 }
}

```

Figura 33: método java usado para realização de testes de ABDR

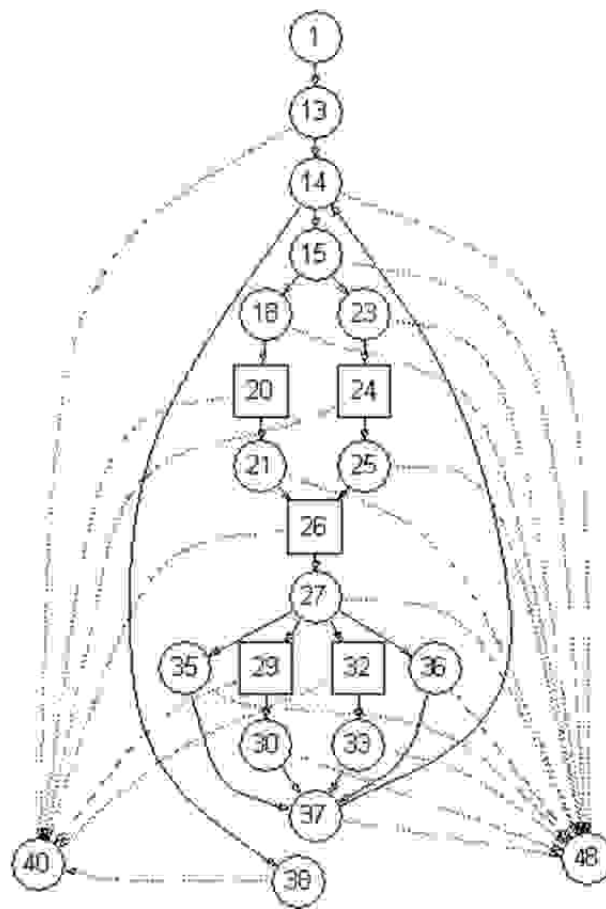


Figura 34: grafo do código da Figura 33



Figura 35: relação de inclusão dos critérios de ABDR para caminhos executáveis. Fonte: (SPOTO, 2000)



Figura 36: relação de inclusão dos critérios de ABDR para caminhos não executáveis. Fonte: (SPOTO, 2000)

5 *Análise de ADR pela JaBUTi*

Este capítulo apresenta as características funcionais e aspectos de implementação relacionados à inclusão do critério todos-t-usos na Ferramenta JaBUTi. A primeira seção apresenta a visão geral das características implementadas na ordem em que foram executadas e divide o projeto em duas fases, estática e dinâmica. A segunda e terceira sessões apresentam as funcionalidades implementadas em cada fase e, em ambas, são detalhados os principais aspectos de implementação.

5.1 Visão Geral das Características Implementadas

As análises realizadas pela Ferramenta JaBUTi podem ser divididas em duas fases, **estática** e **dinâmica**. Na primeira fase, são feitas as análises que independem da execução de um caso de teste, como a criação dos grafos e obtenção dos elementos requeridos. Na segunda fase, são realizadas as análises que dependem da execução dos casos de teste, como a análise de cobertura.

As etapas da implementação do critério todos-t-usos podem, também, ser divididas em duas fases, implementação da fase estática e implementação da fase dinâmica.

5.1.1 Análise Estática

Na fase de análise estática, a inclusão do critério todos-t-usos na Ferramenta JaBUTi requereu que as seguintes características fossem implementadas:

1. No menu de escolha de critérios, a inclusão das opções todos-t-usos-ei (independente de exceção) e todos-t-usos-ed (dependente de exceção), na tela principal;
2. Identificação dos nós de comando SQL relacionados à definição de variável tabela

- e t-uso. Na impossibilidade de identificar os comandos SQL, a ferramenta permite que o comando seja introduzido pelo usuário de teste;
3. Inclusão da lista de variáveis tabela definidas e em uso juntamente aos demais dados fornecidos pela ferramenta para cada nó na interface *def-use graph*;
 4. Apresentação dos nós SQL identificados, em formato diferenciado (nós quadrados) na interface *def-use graph*;
 5. Apresentação da lista de variáveis tabelas sendo definidas no nó de definição de uma associação definição-t-uso, ao se clicar com o botão direito do mouse sobre o respectivo nó, nas interfaces *def-use graph*, *Current Bytecode File* e *Current Source File*;
 6. Identificação dos elementos requeridos e a apresentação na interface *Required Elements*;
 7. Cálculo do peso para cada elemento requerido e a respectiva apresentação, através de uma escala de cores, nas interfaces *def-use graph*, *Current Bytecode File* e *Current Source File*;

5.1.2 Análise Dinâmica

Na fase de análise dinâmica, a inclusão do critério todos-t-usos na Ferramenta JaBUTi requereu que as seguintes características fossem implementadas:

1. Detecção de quais elementos requeridos foram exercitados;
2. Atualização das cores dos pesos;
3. Realização dos cálculos de cobertura para o critério;
4. Apresentação da análise de cobertura;
5. Apresentação de relatório;

Foi apresentada uma visão geral dos requisitos que caracterizam a inclusão do critério todos-t-usos na Ferramenta JaBUTi. As próximas sessões abordam, de forma mais detalhada, as funcionalidades implementadas.

5.2 Características e Aspectos de Implementação na Fase Estática

A implementação do critério todos-t-usos na Ferramenta JaBUTi envolveu diversos aspectos. O primeiro deles foi a inclusão, na interface gráfica, de facilidades que permitam sua utilização, pelo usuário, como mostrado na Figura 37.

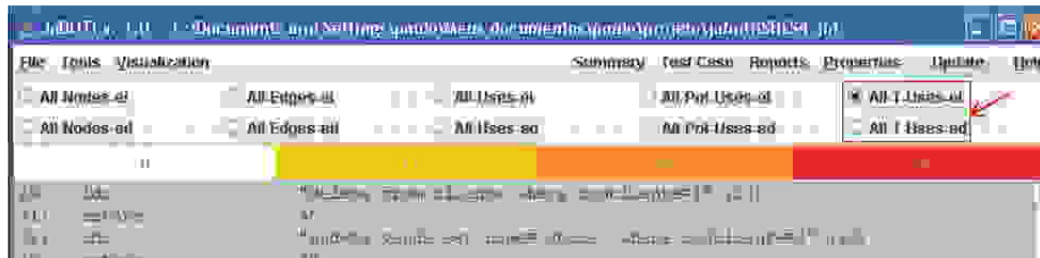


Figura 37: Menu de Critérios

Outro aspecto é a **capacidade de se detectar a presença de uma definição ou t-uso**. Para gerar o grafo de fluxo de dados, a JaBUTi primeiro gera um grafo de instruções (não apresentável para o usuário), e analisa o bytecode instrução a instrução em busca de definições e usos.

Durante esta análise, se for encontrado um método responsável pelo envio de uma expressão SQL para execução, o nó atual será marcado como um nó SQL. Os métodos reconhecidos pela JaBUTi são: *execute()*, *executeQuery()* e *executeUpdate()* das classes *Statement* ou *PreparedStatement*, e o método *commit()* da classe *Connection*. A instrução *INVOKEINTERFACE* invoca um método externo, dentre eles os métodos responsáveis pelo envio da expressão SQL. Desta forma, as seguintes linhas de instruções são consideradas nós SQL:

```
invokeinterface java.sql.Statement.execute(Ljava/lang/String;)
invokeinterface java.sql.Statement.executeQuery(Ljava/lang/String;)
invokeinterface java.sql.Statement.executeUpdate(Ljava/lang/String;)
invokeinterface java.sql.PreparedStatement.execute(Ljava/lang/String;)
invokeinterface java.sql.PreparedStatement.executeQuery(Ljava/lang/String;)
invokeinterface java.sql.PreparedStatement.executeUpdate(Ljava/lang/String;)
invokeinterface java.sql.Connection.commit()
```

Os parâmetros passados, em cada um dos métodos citados, pertencentes à classe *Statement* ou *PreparedStatement*, representam uma expressão SQL. Uma vez determinado que o nó de instrução é um nó SQL, se o parâmetro passado for uma constante, é possível

para a JaBUTi analisar o conteúdo e determinar qual o comando SQL e quais as tabelas envolvidas:

Para determinar o valor da constante passada como parâmetro foi utilizado um vetor já existente que tem como função armazenar dados sobre o estado da máquina virtual em cada nó de instrução e que são utilizados para se determinar o fluxo de dados do grafo. De uma forma geral, é possível determinar uma definição ou uso apenas pela semântica da instrução, como *aload_3*, em que ocorre um uso de uma variável local 3. Mas em determinados casos, faz-se necessário conhecer o estado da máquina virtual no momento em que a instrução é executada, para que seja possível determinar um uso ou definição. Um exemplo desta situação é a definição de um vetor, como *Vetorteste[8]=10*.

Esta atribuição gera as seguintes linhas bytecode:

```
0:  aload_0
1:  getfield      classeteste.Vetorteste [I (2)
4:  bipush       8
6:  bipush       10
8:  iastore
```

A linha 0 coloca o objeto do tipo *classeteste*, que possui o atributo *Vetorteste*, na pilha. A linha 1 carrega na pilha o valor do atributo *Vetorteste*. As linhas 4 e 6 colocam os valores 8 e 10, respectivamente, na pilha. E a linha 8 realiza a atribuição do que está no topo da pilha (valor 10), na posição 8, no campo *Vetorteste*. Desta forma, faz-se necessário conhecer a pilha de execução, pois apenas a linha 8 não indica o que está sendo definido.

Para se determinar o conteúdo de uma constante no momento que o método de chamada é executado, utiliza-se o mesmo conceito. Assim, seja o trecho de programa Java a seguir:

```
String sqlCommand="insert into client values (1,'Maria')";
stmt.execute(sqlCommand);
```

E seu respectivo bytecode:

```
1  ldc  "insert into cliente values (1,'Maria')"(14)
2  astore  \%-5
3  aload_2
4  aload  \%-5
5  invokeinterface java.sql.Statement.execute(Ljava/lang/String;)Z (15)  2 0
```

Para determinar o valor da constante quando a JaBUTI realiza a análise do código e identifica a instrução **LDC**, é enviado para o topo da pilha de operandos (objeto *VMStack*), o tipo e a constante (que pode ou não ser uma expressão SQL) armazenados em um objeto *VMStackElement*. Ao se detectar a instrução **invokeinterface**, a informação do topo da pilha é recuperada, incluindo-se a constante.

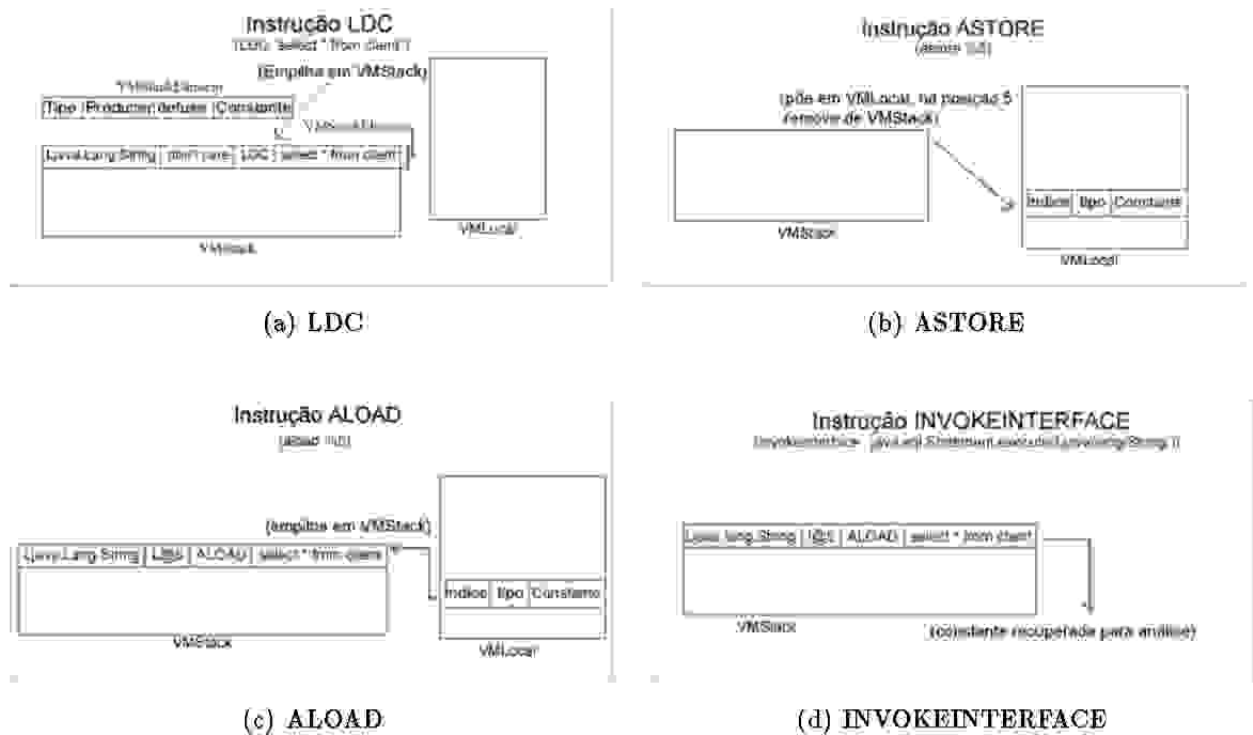


Figura 38: Estados da JVM

Conforme a Figura 38(a), o objeto *VMStackElement*, que possui a informação sobre o tipo e a constante, é enviado ao topo da pilha de operandos (*VMStack*), ao se identificar a instrução **ldc**. Neste exemplo, a constante representa a expressão SQL. Quando identificada a instrução de armazenamento **astore**, o tipo e a constante são enviados a um vetor de variáveis locais (*VMLocal*) na posição identificada por um índice (no exemplo, 5) e, em seguida, *VMStackElement* é removido do topo da pilha (Figura 38(b)).

A seguir, a primeira instrução de carregamento, **aload_2**, refere-se a `java.sql.Statement`. Na segunda instrução de carregamento, **aload %5**, o tipo e a constante localizados na posição 5 do vetor de variáveis locais *VMLocal* são enviados para o topo da pilha de operandos (Figura 38(c)). Ao se identificar a instrução **invokeinterface**, a informação contida no topo da pilha é recuperada (Figura 38(d)). Neste ponto, a constante está disponível para ser avaliada.

Tomando-se como base os formatos propostos em ANSI, ISO e IEC (1999), a avaliação para se determinar o nome da tabela ou tabelas envolvidas é feita primeiramente analisando-se qual o comando SQL, isto é, a primeira palavra da expressão. As seguintes regras são aplicadas: para os comandos *insert*, *update* e *delete*, o nome da tabela é a próxima palavra após *into*, *update* e *from*, respectivamente. Para o comando *select*, o nome da tabela ou tabelas envolvidas é aquela localizada entre *from* e *where* (ou fim da expressão, caso não haja *where*), após uma vírgula entre *from* e *where* e após *join*. Também é observado se existe uma expressão dentro de outra, e se existir, a mesma análise é realizada na *subquery*, ou em expressões como:

```
select * from cliente union all select * from venda
```

Após determinado qual o comando SQL e qual ou quais as tabelas envolvidas, é registrado no nó atual, a presença de t-uso e/ou definição de variável tabela. Originalmente, o objeto *InstructionNode*, isto é, o objeto que representa o nó de instrução, possui um objeto *HashSet* que armazena as variáveis de programa em uso (*localUse*) e um que armazena as variáveis de programa em definição (*localDef*). Utilizando-se o mesmo conceito, foram criados dois novos *HashSets*, **localTUse** e **localTDef**, para se armazenar os t-usos e definições de variável tabela. O valor registrado é o nome da tabela, ou tabelas, presente na expressão SQL.

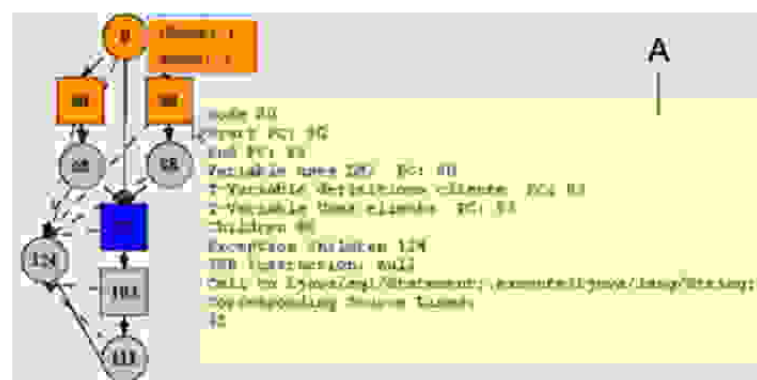


Figura 39: Definição e T-Uso

Sendo um comando *select*, *update*, *delete* ou *insert*, registra-se a presença de t-uso. Caso seja um *update*, *delete* ou *insert*, registra-se, também, a presença de uma definição de variável tabela (conforme explicado no Capítulo 4, Seção 1). A quinta e sexta linhas do quadro A da Figura 39 apresentam um exemplo de uma expressão SQL *insert*, em que ocorre um t-uso e uma definição da variável tabela cliente. Se não for possível determinar a expressão SQL, como em uma situação em que a expressão é passada como parâmetro

através de uma variável, e não uma constante, a cor do nome do nó será cinza, indicando que existe um método de chamada para execução de um comando SQL que não pôde ser analisado (nó SQL indefinido), como na Figura 40.

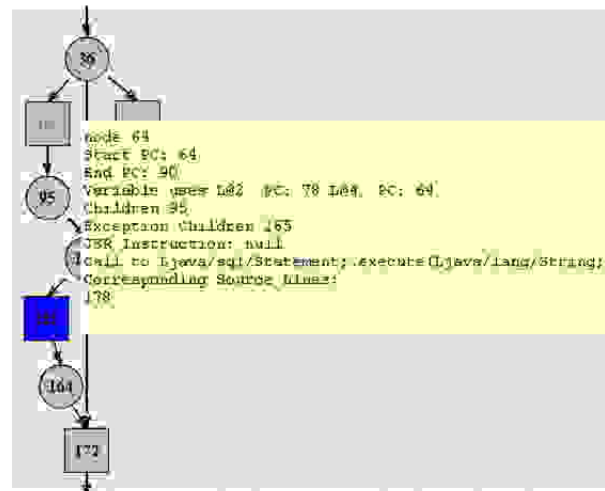


Figura 40: Nó SQL Indeterminado (nome do nó em cinza)

Nestes casos, a ferramenta permite que o usuário entre manualmente com a expressão SQL, clicando-se no nó SQL indefinido (Figura 41).

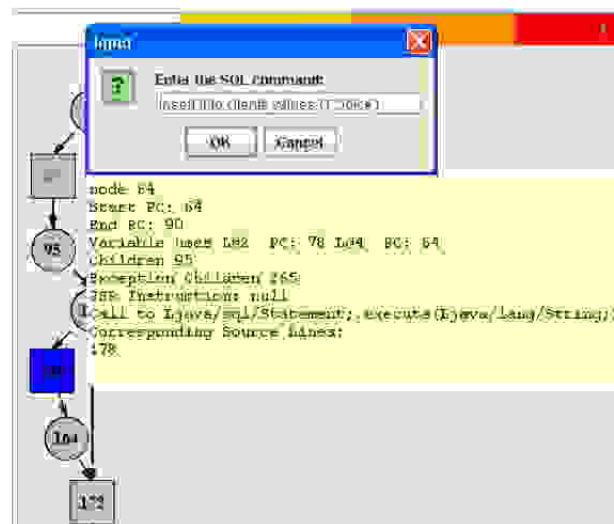


Figura 41: Nó SQL Indeterminado (Entrada Manual de Expressão SQL)

Inserida a expressão, a JaBUTi pede confirmação para atualizar o projeto, dando a flexibilidade para que o usuário possa inserir comandos em outros nós SQL indefinidos, caso existam, e realizar a atualização de uma única vez.

Ao ser atualizado, o mesmo processo de análise sobre a expressão SQL é realizado e é registrada no nó a presença de t-use e definição de variável tabela (Figura 42). O nome

do nó é mantido em cinza para que seja possível o usuário alterar a expressão SQL inserida anteriormente.

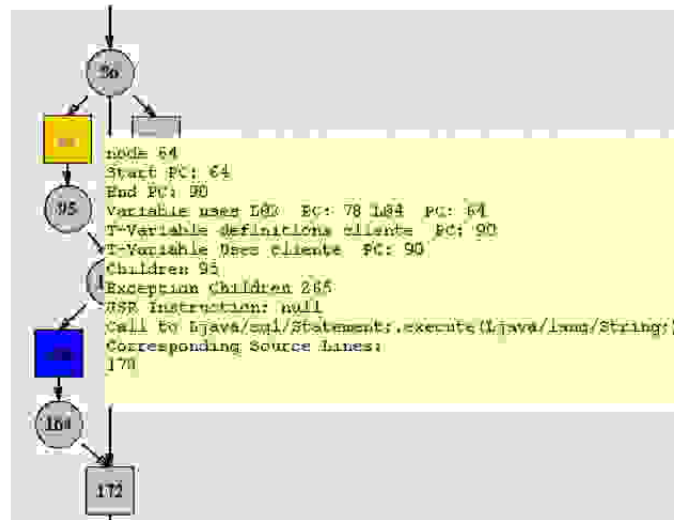


Figura 42: Grafo Atualizado Após Entrada Manual de Expressão SQL

A alteração da representação dos nós SQL para a forma quadrada e a apresentação das variáveis tabela em uso e definição têm início na criação do grafo de fluxo de dados, ao se agrupar as instruções de um mesmo fluxo de controle em um único bloco. A divisão entre estes blocos é feita pela classe *CFG* (Grafo de Fluxo de Controle).

Uma alteração nesta classe foi introduzida para se dividir o bloco após um nó de instrução SQL. Como cada nó de instrução contém informação que o identifica caso seja um nó com comando SQL detectado, com um comando SQL indefinido ou um *commit*, a mesma informação é utilizada durante o momento de representar o grafo visualmente. Será um nó quadrado caso possua uma das três características citadas, e será pintado de azul, se for um nó *commit*.

Uma vez determinados e representados os nós de definição e t-uso, a JaBUTi deve identificar os elementos requeridos. Para tanto, foi aproveitado, inicialmente, o mesmo processo utilizado para identificação de elementos requeridos do critério todos-usos, fazendo-se apenas as adaptações necessárias, como a criação das classes *AllTUses* e *DefTUse*, baseadas nas já existentes *AllUses* e *DefUse*, no pacote *criteria*.

Mantendo o padrão da JaBUTi, o critério todos-t-usos foi dividido em dois, **all-t-usos-ei** (todos-t-usos independentes de exceção) e **all-t-usos-ed** (todos-t-usos dependentes de exceção).

5.2.1 Elementos Requeridos do Critério All-T-Uses-ei

Em um método da classe *AllTUses*, antes de se verificar a presença de associação entre definição e uso de variável tabela, o grafo é analisado em busca de pelo menos um *commit* e são armazenados os nomes de todos os nós que o possuem. Não estando presente, a ferramenta considera que o *commit* é realizado logo após a execução de cada comando SQL. Uma associação em que há definição de uma variável tabela cliente no nó 10 e t-uso em um arco (50,55), por exemplo, teria o seguinte formato, caso não houvesse *commit* no método:

<cliente, <10,nó sucessor>,(50,55)>

O mesmo conceito de detecção de elemento requerido de todos-usos é aplicado, porém, na existência do *commit* no método, será verificado quais nós de *commit* estão presentes entre a definição e o t-uso, e validado o elemento requerido considerando-se, assim, sua persistência.

```

26 if (Inserir)
    {
30     stmt.execute("insert into cliente values (1,'Joyce')");
        System.out.println ("Comando insert executado");
39     switch (Acao)
        {
76         case 1:
            metodo1();
            Con.commit();
            break;
            case 2:
89             metodo2();
                Con.commit();
                break;
            default:
102             metodo3();
        }
106     ResultSet rs=stmt.executeQuery("select * from cliente c inner join
                                     venda v on c.codcliente=v.codcliente");
        } else {
120     Con.commit();
        }
126 System.out.println("Saida");

```

Na Figura 43, é representado o grafo do trecho de código acima. O nó 30 possui uma definição de variável tabela e o nó 106 possui um t-uso da mesma variável tabela (*cliente*). Para se abstrair o elemento requerido, primeiramente identificam-se e armazenam-se todos os *commits* do método (nós 76, 89, 120). Caso não estivessem presentes, a ferramenta consideraria o nó 39 como *commit*. Isto ocorre para permitir que o elemento requerido seja abstraído mesmo quando se programa em modo de *auto-commit*, isto é, situação em que o Java considera que após cada instrução SQL um *commit* é executado implicitamente. Neste cenário hipotético, haveria um único elemento requerido:

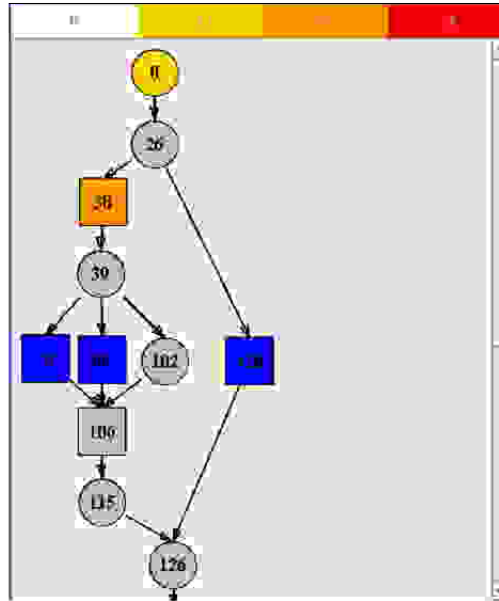


Figura 43: Grafo de Fluxo de Dados de ABDR (Nós commit)

```
<cliente,<30,39>,(106,115)>
```

Um método da classe *AllTUses* determina a relação entre a definição e t-uso de forma semelhante ao critério todos-usos. No exemplo da Figura 43, identifica-se a associação entre o nó 30 e o arco (106,115). Porém, antes de registrar a associação como um elemento requerido, é verificado se, entre a definição e o t-uso, existe um nó de *commit* (cujo nome já foi inicialmente conhecido no início da execução do método). Destes, apenas o 76 e o 89 estão no caminho entre a definição e t-uso. Assim, ao se registrar a associação, o que originalmente seria apenas um elemento requerido se não fossem considerados os *commits*, tornam dois:

```
<cliente,<30,76>,(106,115)>, e;  
<cliente, <30,89>,(106,115)>
```

Além destes elementos requeridos, a JaBUTi considera que, para cada t-uso sem definição, existe uma definição global representada no nó 0 do método, para que se force a execução de um caso de teste que cubra o t-uso. No exemplo, dois novos elementos requeridos são considerados:

```
<venda,<0,0>,(106,115)>, e;  
<cliente, <0,0>,(30,39)>
```

No primeiro caso, existe um t-uso da variável tabela venda (nó 106) que não possui uma definição. No segundo, como a instrução SQL no nó 30, *insert*, implica em um t-uso

seguido de uma definição, este primeiro t-uso não está associado a uma definição em nós anteriores. Para ambas situações, a ferramenta considera a existência de uma definição global persistente para *venda* e *cliente* no nó 0. Uma vez detectado o elemento requerido, este é armazenado em forma de um objeto da classe DefTUse.

5.2.2 Elementos Requeridos do Critério All-T-Uses-ed

O objetivo principal de exercitar um elemento requerido que seja dependente de exceção, sob o ponto de vista do critério todos-t-usos é focar a atenção do usuário de teste para o arco de saída de um t-uso que tem seu fluxo dirigido para um tratamento de exceção, ou ainda, permitir a avaliação do comportamento do programa diante de uma instrução SQL de t-uso mal sucedida. Em um trecho de código:

```

try{
26   stmt.execute("insert into cliente values (1,'Joyce')");
35   Con.commit();
42   ResultSet rs=stmt.executeQuery("select * from cliente c inner join venda v on
                                     c.codcliente=v.codcliente");
51   System.out.println ("Fim");
   } catch (SQLException e){
64   System.out.println("Ocorreu um erro: " + e);
   }

```

Os grafos da Figura 44 representam, respectivamente, o critério todos-t-usos-ei e todos-t-usos-ed. Em ambos, observa-se a definição em 26 e o commit em 35. No grafo da esquerda, é representado o arco de t-uso independente de exceção (42,51) e no da direita, o arco de t-uso dependente de exceção (42,64).

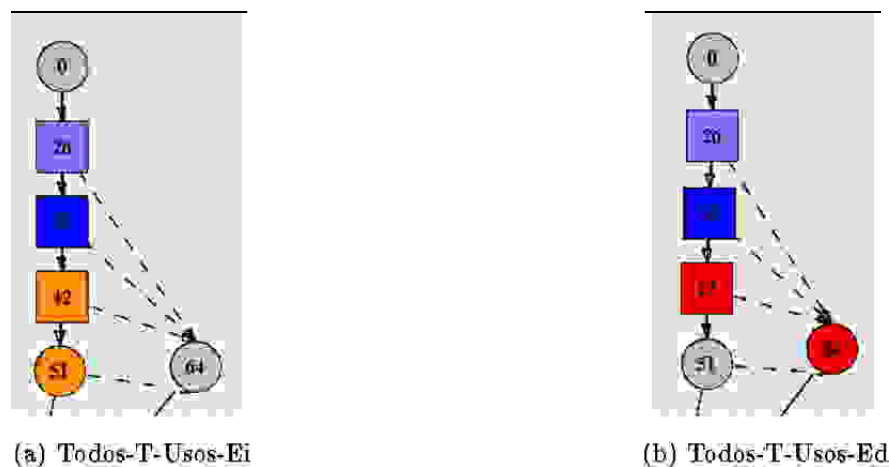


Figura 44: Todos-T-Usos-Ei e Todos-T-Usos-Ed

Além de considerar elementos requeridos que possuam o arco de saída dependente de exceção como mostrado, a ferramenta também deve considerar elementos requeridos que possuam todo o t-uso dentro de um bloco de tratamento de exceção, que é o padrão utilizado no critério todos-usos. No trecho de código e respectivo grafo:

```

try{
26  stmt.execute("insert into cliente values (1,'Joyce')");
35  Con.commit();
   System.out.println ("Fim");
} catch (SQLException e){
53  ResultSet rs=stmt.executeQuery("select * from cliente c inner join venda v on
                                   c.codcliente=v.codcliente");
64  System.out.println("Ocorreu um erro: " + e);
}

```

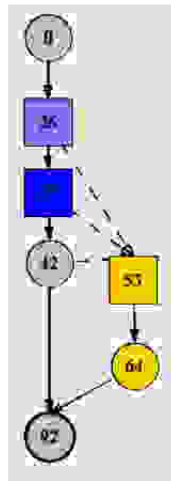


Figura 45: Todos-T-Usos-Ed (t-uso dentro de bloco de exceção)

Há um arco de t-uso cujos nós estão todos dentro do bloco *catch*. Neste cenário, há a presença do elemento requerido $\langle \text{cliente}, \langle 26, 35 \rangle, (53, 64) \rangle$, como mostrado na Figura 45.

Uma vez detectado o elemento requerido, assim como no critério todos-t-usos-ei, este é armazenado em forma de um objeto da classe *DefTUse*.

5.2.3 Distribuição de Pesos

A distribuição de peso segue o conceito apresentado no Capítulo 3, Seção 1.

O valor do peso, representado por uma escala de cores, indica quantos elementos requeridos obrigatoriamente serão exercitados, caso um determinado elemento requerido seja exercitado.

A representação dos pesos é feita de duas formas. Como padrão, são pintados apenas os nós de definição dos elementos requeridos. Supondo que o nó de definição colorido pertença a mais de um elemento requerido, este terá o valor do elemento requerido de maior peso.

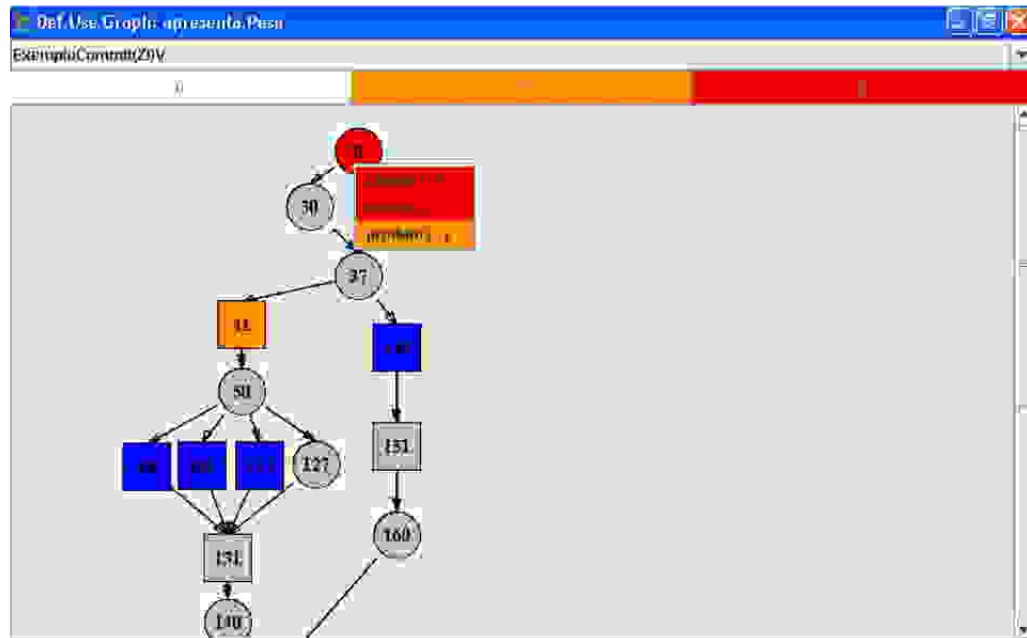


Figura 46: Distribuição de Pesos Representada em Definições

Na Figura 46 é representado um trecho de código que possui arcos de t-uso em (151,160), dada a presença da expressão *select * from produto*, e (41,50), devido à expressão *insert into cliente values (1,'Joice')*. Há, também, um arco de t-uso em (131,140), pela presença da expressão *select * from cliente c inner join venda v on c.codcliente=v.codcliente*, sendo este o único t-uso associado a uma definição persistente ($\langle \text{cliente}, \langle 41, 88 \rangle, (131, 140) \rangle$, $\langle \text{cliente}, \langle 41, 101 \rangle, (131, 140) \rangle$ e $\langle \text{cliente}, \langle 41, 114 \rangle, (131, 140) \rangle$).

Aqui existem três t-usos que não possuem uma definição para serem associadas:

- 2.1 Arco (41,50), com respeito a cliente;
- 2.2 Arco (151,160), com respeito a produto, e;
- 2.3 Arco(131,140), com respeito a venda (mas possui associação com a definição em 41, com respeito a cliente, como listado anteriormente);

A ferramenta considera uma definição persistente e global representada no nó 0

para cada t-uso que não possuir uma definição para ser associada, como mostrado na Figura 46, em que as três variáveis tabelas são listadas no nó 0.

Observa-se que cada uma das variáveis está pintada de uma cor: *Cliente* e *venda* estão pintadas de vermelho, representando o peso 2, e *produto* está pintado de laranja, representando o peso 1.

Isto significa que, dos elementos requeridos que envolvem *cliente* e dos elementos requeridos que envolvem *venda*, o de maior peso tem valor 2. E dos elementos requeridos que envolvem *produto*, o de maior peso tem valor 1.

Mas apenas por este grafo, não é possível determinar quantos elementos requeridos existem para as três variáveis tabelas citadas, pois os arcos de t-uso não estão evidenciados. Para que se possa visualizar com mais detalhe os elementos requeridos, deve-se escolher uma das variáveis tabelas.

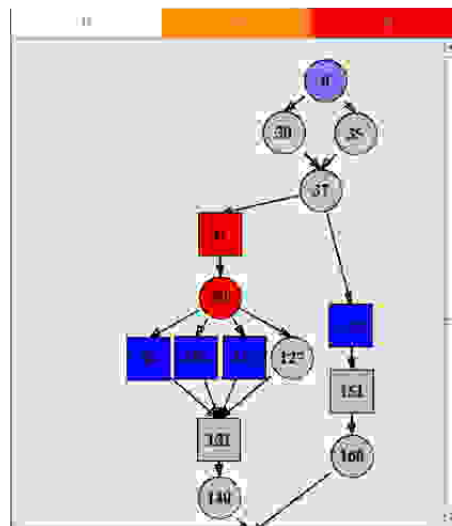


Figura 47: Distribuição de Pesos Representada em T-Usos

Na Figura 47, foi escolhida a variável tabela *cliente*. O nó de definição tem sua cor alterada para azul e os nós de t-uso são pintados com a cor de seus respectivos pesos. Neste exemplo, existe apenas um t-uso para a definição do nó 0 com respeito a *cliente*. A cor vermelha representa o valor 2. Isto indica que se o elemento requerido $\langle \text{cliente}, \langle 0,0 \rangle, (41,50) \rangle$ for exercitado, pelo menos dois elementos requeridos serão exercitados. Estes elementos requeridos são:

```
<cliente, <0,0>, (41,50)>, 2;
<venda, <0,0>, (41,50)>.
```

É possível notar que a ferramenta não considerou os elementos requeridos:

```

<cliente, <41,88>, (131,140)>;
<cliente, <41,101>, (131,140)>;
<cliente, <41,114>, (131,140)>, e;
<produto, <0,0>, (151,160)>.

```

Tais elementos não foram considerados porque não serão necessariamente exercitados, ou nunca serão exercitados, caso $\langle \text{cliente}, \langle 0,0 \rangle, (41,50) \rangle$ seja exercitado. A implementação seguiu o padrão utilizado para o critério todos-usos, com uma alteração para considerar o nó *commit*. Na Figura 47, se o *commit* não fosse considerado, o peso do arco $(41,50)$, que no exemplo é 2, seria 5 pois os elementos requeridos a seguir seriam considerados:

```

<cliente, <41,88>, (131,140)>;
<cliente, <41,101>, (131,140)>;
<cliente, <41,114>, (131,140)>.

```

Isto se daria porque o algoritmo consideraria apenas os nós 41, 131 e 140. Estes, por sua vez, são nós que obrigatoriamente seriam exercitados caso o elemento requerido $\langle \text{cliente}, \langle 0,0 \rangle, (41,50) \rangle$ fosse exercitado, implicando em mais três elementos requeridos obrigatoriamente exercitados. Mas como os nós de *commit* integram os três elementos requeridos, os elementos em questão não podem ser considerados como "obrigatoriamente exercitados", pois um caso de teste poderia exercitar o caminho:

```

<0, 30, 37, 41, 50, 88,131,140>

```

E não passar pelos elementos requeridos $\langle \text{cliente}, \langle 41,101 \rangle, (131,140) \rangle$ e $\langle \text{cliente}, \langle 41,114 \rangle, (131,140) \rangle$. Ou exercitar o caminho:

```

<0, 30, 37, 41, 50, 101,131,140>

```

E não passar pelos elementos requeridos $\langle \text{cliente}, \langle 41,88 \rangle, (131,140) \rangle$ e $\langle \text{cliente}, \langle 41,114 \rangle, (131,140) \rangle$. Ou exercitar o caminho:

```

<0, 30, 37, 41, 50, 114,131,140>

```

E não passar pelos elementos requeridos $\langle \text{cliente}, \langle 41,88 \rangle, (131,140) \rangle$ e $\langle \text{cliente}, \langle 41,101 \rangle, (131,140) \rangle$. Ou exercitar o caminho:

```

<0, 30, 37, 41, 50, 127,131,140>

```

E não passar por nenhum dos três elementos requeridos em questão.

5.3 Características e Aspectos de Implementação na Fase Dinâmica

Uma vez implementadas as funcionalidades da fase estática, a JaBUTi deve possuir a capacidade de detectar quais elementos requeridos foram exercitados em cada caso de teste.

Para tanto, utilizou-se o mesmo algoritmo aplicado ao critério todos-usos, porém foi realizada uma alteração para que o nó de *commit* fosse considerado. Inicialmente, após a execução de um caso de teste, o caminho exercitado é capturado e passado como parâmetro na chamada de um método da classe *AllUses* que irá determinar quais elementos requeridos foram exercitados.

Originalmente, na classe *AllUses*, busca-se todos os elementos requeridos e, um a um, determina-se se a definição e o uso estão inseridos no caminho. Primeiro o caminho é percorrido em busca da definição. Se encontrada, procura-se pelo uso a partir do nó seguinte ao de definição.

A alteração no método da classe *AllUses* inicia-se a partir do ponto em que definição e t-uso foram encontrados dentro do caminho. Em vez do elemento requerido ser considerado exercitado, é realizada uma busca entre o nó de definição e o de t-uso em busca do nó de *commit*. Na Figura 48(a) é mostrado o grafo de um método após ter sido executado um caso de teste que exercitou o caminho: $\langle 0, 36, 64, 95, 153, 158, 164, 172, 181 \rangle$

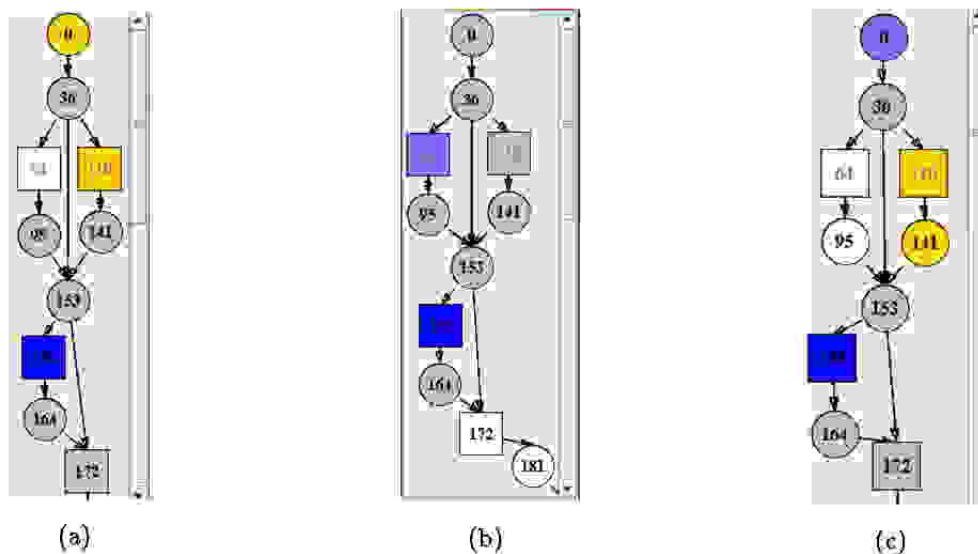


Figura 48: Representações de Cobertura de Elemento Requerido

O método possui um comando *insert* no nó 64 e um comando *delete* no nó 110. Existem quatro elementos requeridos, como pode-se observar na Figura 49.

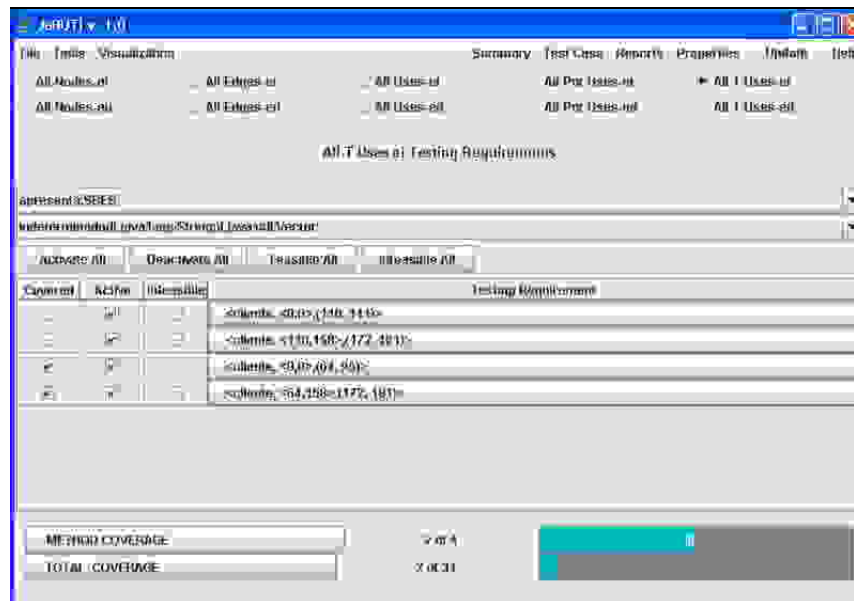


Figura 49: Interface de Elementos Requeridos

Seguindo o padrão de representação de pesos através de cores, em um primeiro momento, o grafo é atualizado, descontando-se todos os elementos requeridos que foram exercitados ao se calcular os pesos. É apresentada em branco, primeiramente, a definição do elemento requerido exercitado, ou elementos requeridos exercitados.

Ao se escolher o nó de definição 64, o esquema de cores é alterado, evidenciando os t-usos relacionados à definição (Figura 48(b)). Se escolhido o nó 0, a escala de cores é alterada como na Figura 48(c).

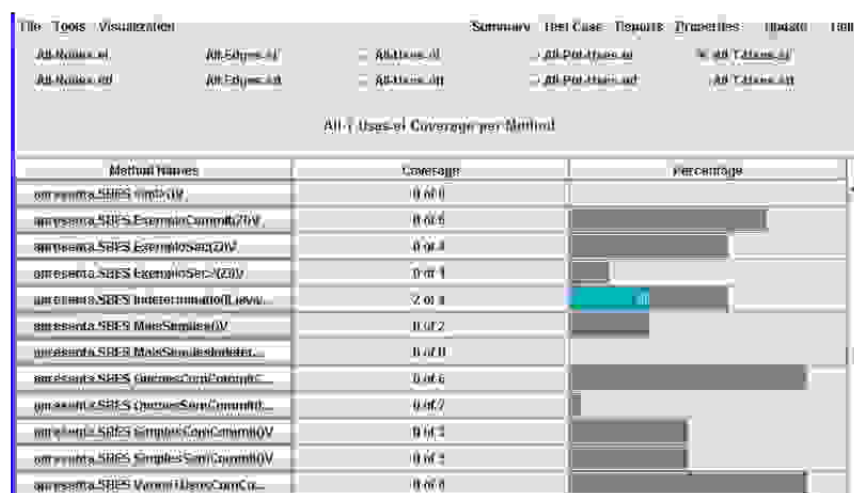


Figura 50: Análise de Cobertura por Método

Tendo-se implementada a capacidade de se determinar quais elementos foram exercitados, a etapa final foi **incluir os dados referentes ao critério todos-t-usos-ei e todos-t-usos-ed nos relatórios**. O relatório pode ser visualizado tanto pela própria JaBUTi como gerado em HTML.

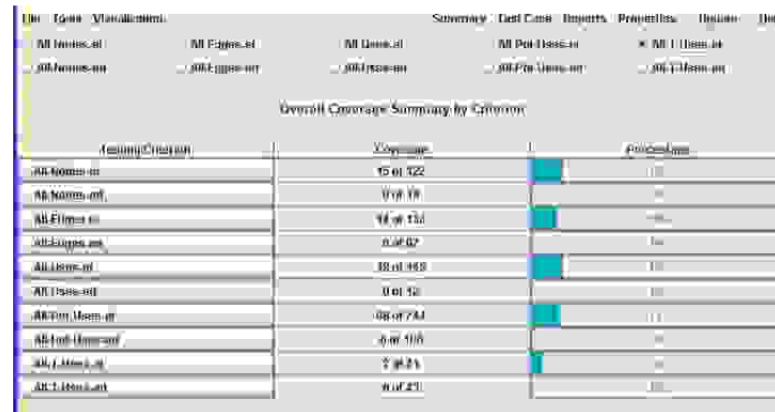


Figura 51: Análise de Cobertura por Critério

Pela JaBUTi, pode-se obter informação, por classe, método ou critério. Por classe e método, são mostrados os dados referentes apenas ao critério escolhido, como na Figura 50. E por critério, como na Figura 51.

Os relatórios gerados em HTML podem ser por critério, método ou classe, como apresentado na JaBUTi, ou um relatório mais completo, com dados sobre cada método, classe e critério, incluindo-se quais elementos requeridos foram cobertos e não cobertos.

5.4 Conclusão

Este capítulo abordou os principais aspectos de funcionalidade e implementação envolvidos na inclusão do critério todos-t-usos na Ferramenta JaBUTi. Foi apresentada uma visão geral sobre as funcionalidades na ordem em que foram implementadas.

O projeto foi dividido em duas fase, implementação da fase estática e dinâmica. Na primeira fase, concentrou-se esforços em dar à ferramenta a capacidade de detectar nós de definição e t-uso, determinar os elementos requeridos para cada método, numerar suas prioridades em relação ao número de elementos requeridos que serão exercitado caso um em particular o seja, através de um sistema de pesos, e a apresentação destas informações na interface.

Na segunda fase, a ferramenta adquiriu a capacidade de analisar quais elementos

requeridos foram exercitados após a execução de um ou mais casos de teste, realizar a análise de cobertura e apresentar tais informações tanto na interface gráfica da ferramenta quanto em relatórios gerados em HTML.

6 *Estudo de Caso*

Este capítulo apresenta um exemplo prático do uso da ferramenta JaBUTi ao se aplicar os critérios todos-t-usos-ei e todos-t-usos-ed implementados, em um método de classe. Foi utilizado um Sistema de Cadastros de Loja de Cosméticos e o método Java (Apêndice B), alvo do estudo, realiza o cadastro e alteração de um funcionário.

O objetivo principal deste capítulo é apresentar a aplicação da JaBUTi em todas suas etapas, considerando exclusivamente o critério todos-t-usos, e evidenciar as vantagens do uso desta ferramenta.

A primeira sessão apresenta a descrição do Sistema e do método Java a ser testado. A segunda sessão aborda a metodologia a ser empregada. A terceira sessão apresenta a execução do teste e a quarta sessão, a conclusão.

6.1 **Descrição do Sistema**

O Sistema testado é uma aplicação de cadastro de clientes, funcionários, fornecedor, produtos, vendas e compras, para uma loja de cosméticos. Optou-se pela escolha de um Sistema simples, com funcionalidades básicas, para que o foco seja direcionado à mecânica do teste, isto é, aos passos a serem realizados na ferramenta JaBUTi para que os elementos requeridos dos critérios implementados sejam exercitados. Uma pesquisa mais completa sobre a eficácia dos critérios de ABDR, utilizando o mesmo Sistema, pode ser encontrada em Becari (2006).

A página principal do Sistema possui a interface mostrada na Figura 52.

Na barra de menu, todos os itens, com exceção a Arquivo, possuem pelo menos as quatro funcionalidades mostradas em *funcionário* (Figura 52): cadastrar, remover, procurar, alterar. O banco de dados utilizados pela aplicação é MySQL 4.1.11, com



Figura 52: Tela Principal do Sistema de Cosméticos

motor de armazenamento com suporte a transação, InnoDB.

Para a realização do teste, foi escolhido o método responsável pelo cadastramento e alteração dos dados do funcionário. Desta forma serão apresentadas, a seguir, as interfaces e funcionalidades envolvidas.

Escolhida a opção Cadastrar, em Funcionário, na barra de menu principal, abre-se uma tela como a mostrada na Figura 53.

A imagem mostra a interface de cadastro de um funcionário. Ela contém campos de entrada para CPF, RG, Nome, Endereço, Bairro, Data de Nascimento, Salário, Causa, Causa, Estado, CEP e Sexo. No canto inferior direito, há um botão 'Procurar'. No canto inferior central, há dois botões: 'Cadastrar' e 'Cancelar'.

Figura 53: Tela de Cadastramento de Funcionário

Os campos *CPF*, *RG*, *Nome*, *Endereço*, *Bairro*, *Data de Nascimento* e *Salário* não podem estar vazios para que o cadastro seja efetuado. O CPF não pode ser igual ao

de outro funcionário. Ainda, é feita a validação deste campo e caso não esteja correto, o Sistema alerta o usuário e pede confirmação se deve ou não ser cadastrado, abrindo a possibilidade de se cadastrar um CPF inválido.

Para que os dados do funcionário sejam alterados, é apresentada a primeira tela, em que como mostrado na Figura 54.

Pesquisando Funcionarios

Procurar por:

Por:

CPF	Nome	Salário	Endereço	Bairro	Cidade
11111111111	José Almeida	1000.00	Rua dos Almoeiros	Freixo	São Paulo
22222222222	Maria Silva	1500.00	Rua das Flores	Centro	São Paulo
33333333333	João Silva	1200.00	Rua Oswaldo Cruz	Centro	São Paulo

Figura 54: Lista de Funcionários Cadastrados

O usuário pode escolher um entre os funcionários cadastrados e alterá-lo em uma nova tela (Figura 55).

CPF:

RG:

Nome:

Endereço:

Bairro:

Telefone:

Nascimento:

Salário:

Celular:

Cidade:

E-mail:

CEP:

Sexo:

Figura 55: Dados do Funcionário a Alterar

As mesmas regras para cadastrar devem ser utilizadas para a alteração, sendo obrigatórios os campos *CPF*, *RG*, *Nome*, *Endereço*, *Bairro*, *Data de Nascimento* e *Salário*.

6.1.1 Aplicação do Critério Todos-T-Usos com a JaBUTi

Esta sessão apresenta um estudo de caso, abordando todos os passos que devem ser realizados pela ferramenta durante a execução dos casos de teste. O método de classe a ser analisado e testado é *evento_onClick_jButtonCadastrar()*, responsável pelo cadastramento e alteração dos dados do funcionário.

O primeiro passo é criar um novo projeto de teste na ferramenta. Para isto, escolhe-se a classe principal (main), como na Figura 56.

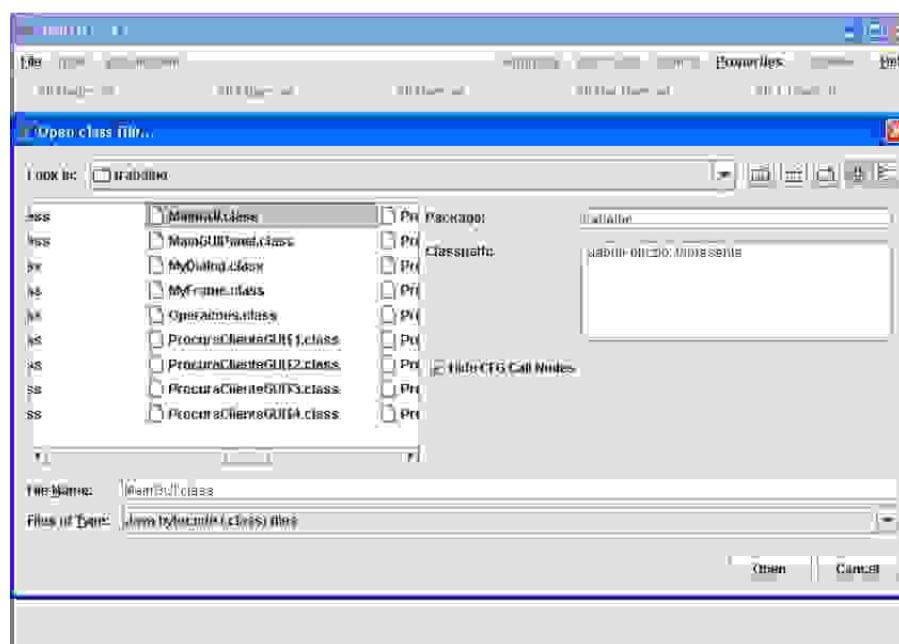


Figura 56: Tela de Escolha da Classe Principal

Selecionada a classe principal, uma nova interface apresentará as classes pertencentes ao Sistema (Figura 57).

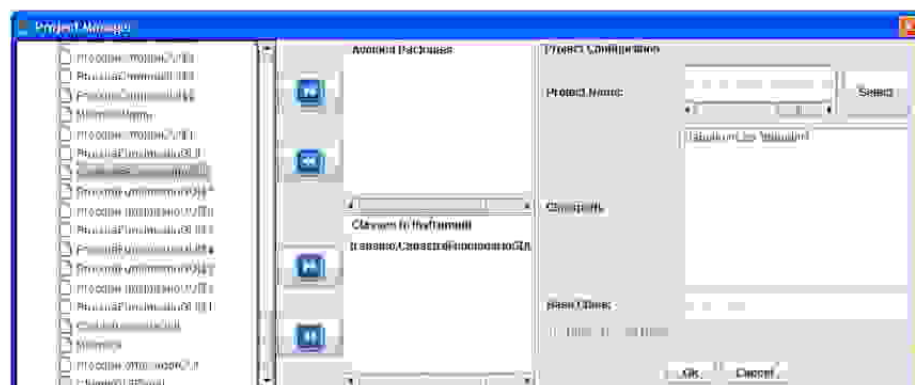


Figura 57: Tela de Escolha das Classes a se Testar

Escolhe-se a classe que possui o método a ser testado (pode ser escolhido mais de um método em uma ou mais classes). Dá-se um nome ao projeto de teste, no estudo de caso, *cosmetico.jbt*, de forma que estes passos não precisarão ser realizados novamente.

O projeto será aberto e, na tela principal, será mostrado o *bytecode* da classe. Como o código fonte está presente, este também pode ser apresentado. O grafo do método a ser chamado pode ser visualizado através da opção **Def-Use Graph**, em *Visualization*, no menu principal.

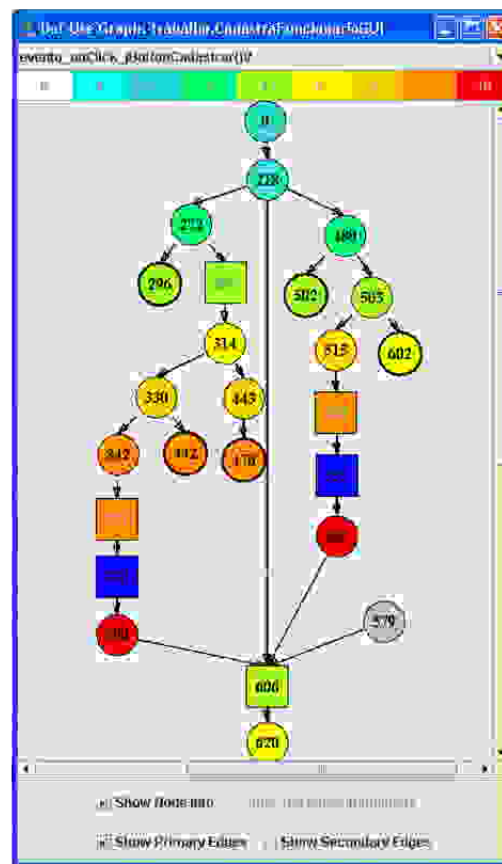


Figura 58: Grafo do Método *evento_onClick_jButtononCadastrar()*

Observando-se o grafo de fluxo de dados com arestas independentes de exceção (Figura 58) é possível notar que a partir do nó 218, o fluxo é dividido para três arcos de saída. Dentre estes, um fluxo pode seguir um caminho que contenha dois comandos SQL (nós 297 e 371) seguidos de um *commit* (380). Outro fluxo pode seguir um caminho que contenha um comando SQL (544) seguido de *commit* (553). Ambos os fluxos convergem para um outro comando SQL (606). O terceiro fluxo segue direto para o nó 606, sendo este o único comando SQL deste caminho.

Através da análise do código, e tendo-se a visão geral fornecida pelo grafo, observa-

se que o método é responsável pelo cadastramento e alteração dos dados do funcionário. E que a divergência do fluxo de dados no nó 218 separa ambas as ações.

Dos nós detectados como comandos SQL, 380 e 553 são *commit*. A ferramenta foi capaz de determinar a presença de definição e/ou t-uso no nó 606, pois a expressão SQL foi passada como uma constante, como pode-se observar na Figura 59.

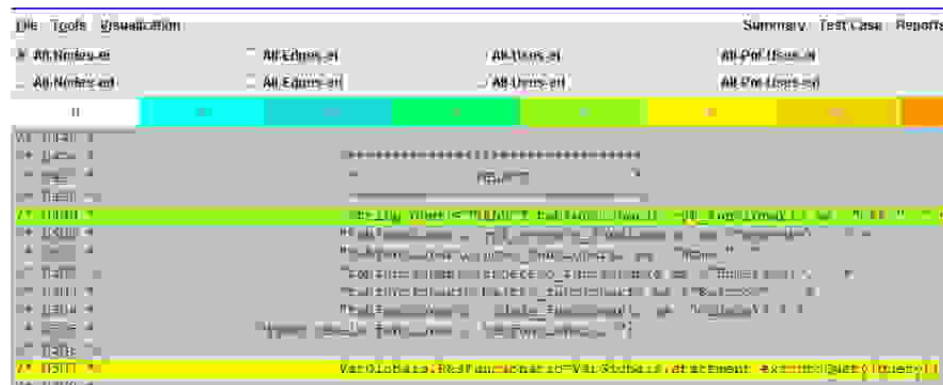


Figura 59: Constante SQL Analisada pela JaBUTi

Na Figura 60 são apresentados os dados relativos ao nó 606.

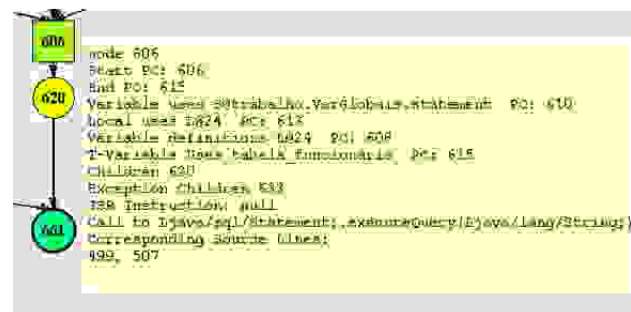


Figura 60: Dados da Análise da Expressão SQL

Neste caso é possível, apenas observando os dados extraídos do nó 606, compreender que se trata de um comando *select*, já que foi detectado apenas um t-uso da variável **tabela_funcionario**.

Em relação aos nós SQL 297, 371 e 544, os comandos SQL não estão presentes em forma de constante. São variáveis e, portanto, deve-se analisar o código em busca de seus valores.

Na Figura 61, é mostrado que o comando SQL a ser executado na linha 416 está contida em uma variável que, por sua vez, receberá o retorno da chamada do método *select117()* de um objeto *QueryBD*. Analisando-se este método foi possível determinar

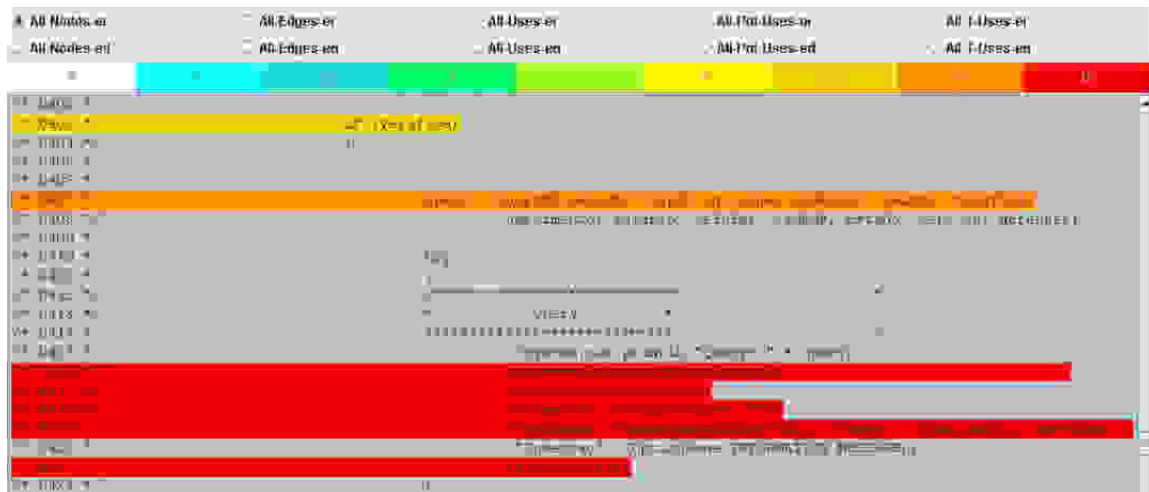


Figura 61: Variável Contendo Expressão SQL (Não Analisável pela JaBUTi)

qual expressão SQL será executada. O mesmo procedimento foi feito para os três nós que a JaBUTi não pôde identificar as expressões, e foi possível determina que:

- O nó 297 executa:

```
"SELECT * FROM tabela_funcionario tabfuncionario
WHERE tabfuncionario.cpf_funcionario = " + cpf + "";
```

- O nó 371 executa:

```
"INSERT INTO tabela_funcionario VALUES (" + cpf + ", " + cpf_gerente + ", " +
  rg + ", " + nome + ", " + endereco + ", " + bairro + ", " + telefone +
  ", " + nascimento + ", " + salario + ", " + celular + ", " + cidade +
  ", " + estado + ", " + cep + ", " + cpf_gerente + ")";
```

- O nó 544 executa:

```
"UPDATE tabela_funcionario SET cpf_funcionario = " + cpf + ",
  cpf_gerente_funcionario = " + cpf_gerente + ", rg_funcionario = " + rg +
  ", nome_funcionario = " + nome + ", endereco_funcionario = " + endereco +
  ", bairro_funcionario = " + bairro + ", telefone_funcionario = " +
  telefone + ", nascimento_funcionario = " + nascimento + ",
  salario_funcionario = " + salario + ", celular_funcionario = " + celular +
  ", cidade_funcionario = " + cidade + ", estado_funcionario = " + estado +
  ", cep_funcionario = " + cep + " " + "WHERE cpf_funcionario = " + cpf +
  "";
```

Em que todos os valores a serem alterados representam variáveis.

Na tese de Spoto (2000), a definição do critério todos-t-usos exige que a mesma tupla exercitada na definição persistente deve ser exercitada no t-uso. Com os três exemplos dos nós apresentados, fica claro que é impossível determinar em uma análise estática, os valores que serão inseridos ou alterados na tabela. Isto apenas é possível em uma análise dinâmica, no momento em que o usuário entrar com os dados a serem manipulados. Esta capacidade poderá ser alvo de projetos futuros. No atual estágio, a verificação

da presença da mesma tupla na definição persistente e no t-uso deverão ficar em cargo do usuário de teste.

Para a ferramenta, como ainda não é possível analisar dinamicamente os dados a serem manipulados, não importa os valores destes dados. O que importa é que a JaBUTi tenha o trecho da expressão SQL suficiente para determinar qual, ou quais, a tabela envolvida. Sendo assim, bastaria saber os seguintes trechos de expressão, para seus respectivos nós:

- Nó 297: "SELECT * FROM tabela_funcionario tabfuncionario WHERE"
- Nó 371: "INSERT INTO tabela_funcionario VALUES "
- Nó 544: "UPDATE tabela_funcionario SET"

Com estes trechos, a JaBUTi é capaz de determinar o fluxo de dados das variáveis tabelas. Tais trechos serão inseridos em seus respectivos nós, como na Figura 62.



Figura 62: Inserção Manual de Expressão SQL

Após atualizado o projeto, a ferramenta passará a considerar a informação contida no trecho de expressão SQL. No exemplo, a Figura 63 apresenta a variável **tabela_funcionario** da expressão inserida (Figura 62).

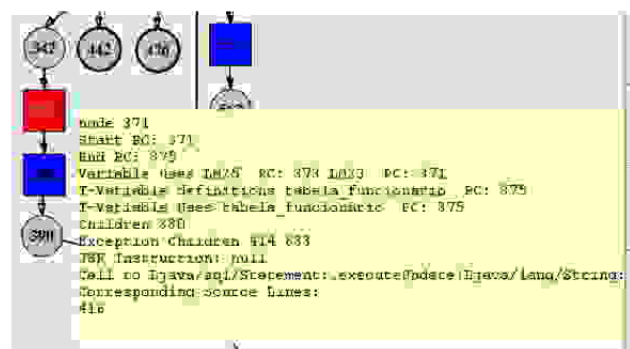


Figura 63: Análise da JaBUTi sobre Expressão SQL Inserida Manualmente

Active All			
Disabled All			
Feasible All			
Infeasible All			
Covered	Active	Infeasible	Feeding Requirement
☐	15	☐	<tabela_funcionario, <010,531,553>
☐	16	☐	<tabela_funcionario, <015,371,300>
☐	17	☐	<tabela_funcionario, <544,553>, <606,620>
☐	18	☐	<tabela_funcionario, <371,388>, <606,620>
☐	19	☐	<tabela_funcionario, <010>, <207,314>
MÉTODO COVERAGE			
TOTAL COVERAGE			

Figura 64: Elementos Requeridos do Método de Teste

E identificará os elementos requeridos, como mostrado na Figura 64.

Até o presente momento do teste, a ferramenta forneceu meios que facilitam a etapa de teste da ABDR, através da criação do grafo com a representação diferenciada dos nós de comando SQL, da visualização do código e as referências das respectivas linhas que cada nó contém, e da análise sobre o fluxo dos dados de variáveis tabelas, suas associações e os elementos requeridos do critério todos-t-usos.

A partir deste ponto, o usuário de teste deverá ter conhecimento suficiente para planejar a execução dos casos de teste. Como citado, a mesma tupla envolvida na definição persistente deve estar presente no t-uso. No método em questão, o t-uso no arco (606,620) significa a busca de todas as tuplas da tabela `tabela_funcionario`, implicando que certamente a tupla inserida <544,553> ou alterada <371,300> também será exercitada no t-uso em questão.

Uma vez escolhidas as entradas a serem utilizadas, inicia-se a execução dos casos de teste. Para efeito de demonstração, o Sistema será executado duas vezes para que se possa visualizar os passos realizados. No primeiro caso de teste será feito um cadastramento de funcionário. No segundo caso de teste será feita uma alteração de funcionário. A linha de comando abaixo foi utilizada para que a JaBUTi possa rastrear a execução do Sistema.

```
java -cp ".;Jabuti-bin.zip;ojdbc14.jar" probe.ProberLoader -P cosmetico.jbt
-cp ".;.\trabalho;ojdbc14.jar" trabalho.MainGUI
```

Para o primeiro caso de teste, efetuou-se um cadastro com os seguintes dados:

Executada a ação de cadastramento, deve-se atentar ao resultado. A simples execução do caso de teste não tem significado se o resultado obtido e esperado não forem comparados.

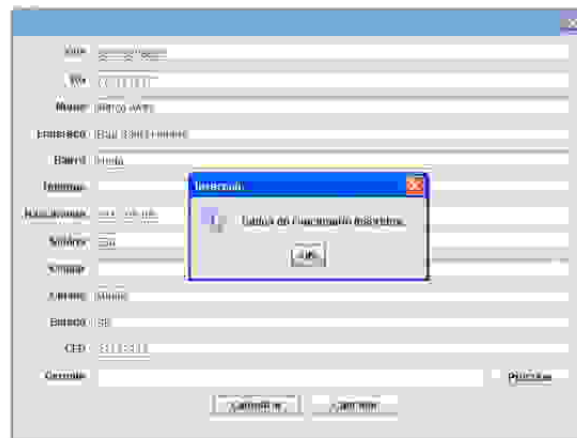


Figura 65: Inserção de Funcionário Concluída com Êxito

Analizados os resultados, ao se selecionar o botão *Update* os elementos requeridos exercitados serão marcados como na Figura 66.

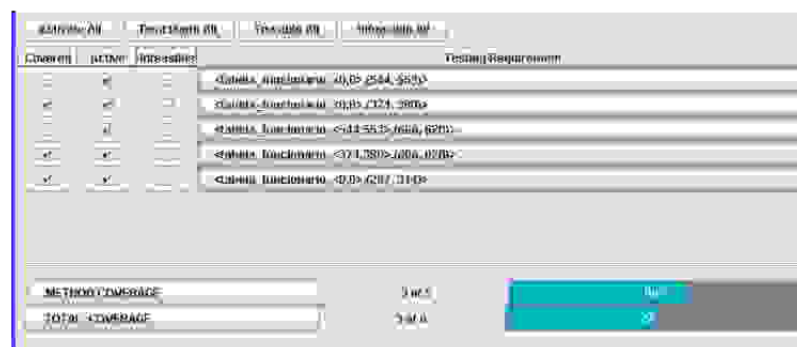


Figura 66: Primeiro Caso de Teste

A análise de cobertura e as cores dos nós serão alteradas, como mostram as Figuras 66 e 67.

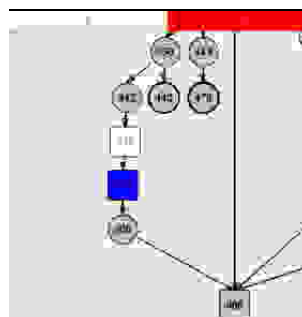


Figura 67: Alteração do Grafo Após Primeiro Caso de Teste

O mesmo processo foi aplicado para a ação de alterar, cobrindo-se todos os elementos requeridos de todos os usuários. Porém, a tela seguinte a de atualização, que apresenta todos os funcionários, mostra no campo *Gerente*, o mesmo CPF do funcionário

(Figura 68).

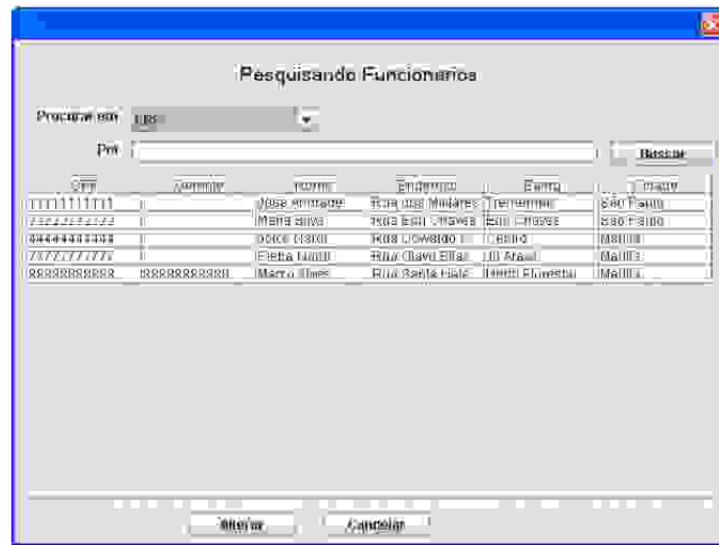


Figura 68: Resultado Diferente do Esperado

A apresentação dos funcionários se dá através do t-uso no arco (606,620), em que há a busca de todos os funcionários da tabela `tabela_funcionario`. Assim, o exercício do elemento requerido $\langle \text{tabela_funcionario}, \langle 544, 553 \rangle, (606, 620) \rangle$ detectou a existência de um defeito.

Em seguida, devem ser realizados os testes do critério todos-t-usos-ed. Na Figura 69, é apresentado o grafo, considerando-se o fluxo de dados dependente de exceções.

Através da análise sobre este grafo, é possível notar que todos os nós, à partir de 218, estão dentro de um bloco de tratamento de exceção (*try*) e caso haja um erro, o fluxo será convergido para o nó 633. Porém, observa-se que alguns nós possuem divergência para dois nós de tratamento de exceção. Isto ocorre porque dentro do bloco de tratamento de exceção maior, existem trechos que estão dentro de outro bloco de tratamento de exceção, isto é, há a presença de um bloco (*try*) dentro de outro. Estes nós são os de comando SQL em que existe a definição de variável tabela (inserção ou alteração). Assim, espera-se que haja um número maior de elementos requeridos para o critério dependente de exceção do que para o independente de exceção. Tais elementos são:

- $\langle \text{tabela_funcionario}, \langle 0, 0 \rangle, (544, 633) \rangle$
- $\langle \text{tabela_funcionario}, \langle 0, 0 \rangle, (371, 633) \rangle$
- $\langle \text{tabela_funcionario}, \langle 0, 0 \rangle, (544, 579) \rangle$

- $\langle \text{tabela_funcionario}, \langle 0,0 \rangle, (371, 414) \rangle$
- $\langle \text{tabela_funcionario}, \langle 544, 553 \rangle, (606, 633) \rangle$
- $\langle \text{tabela_funcionario}, \langle 371, 380 \rangle, (606, 633) \rangle$
- $\langle \text{tabela_funcionario}, \langle 0,0 \rangle, (297, 633) \rangle$

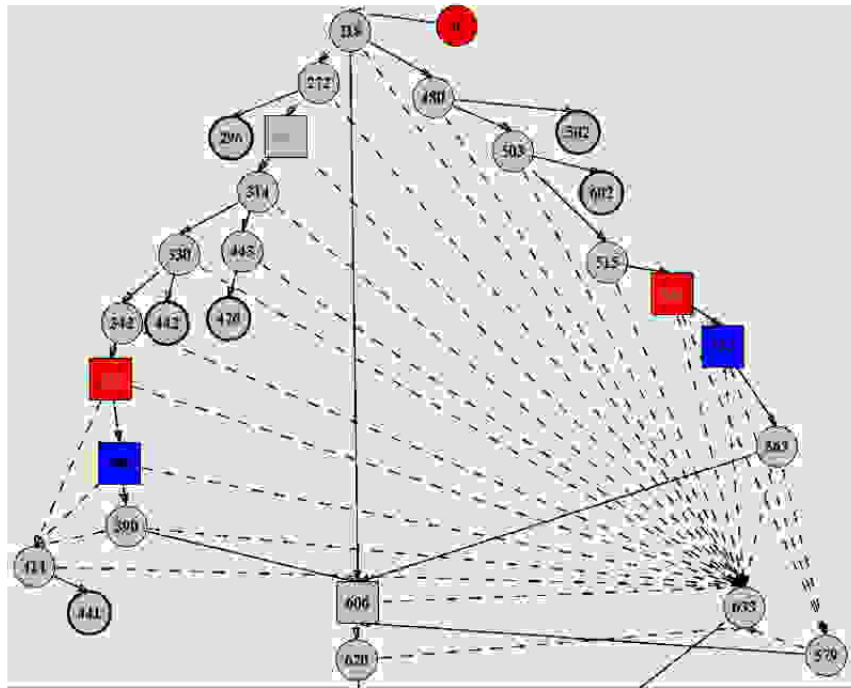


Figura 69: Grafo com Arestas Dependentes de Exceção

Destes elementos, os dois primeiros são inactiváveis, pois se ocorrer uma exceção, o fluxo será direccionado para o nó 414 no caso do primeiro elemento requerido e 579 no caso do segundo elemento requerido. Isto se dá porque os nós 371 e 544 estão dentro de blocos *try* internos a um outro bloco *try*, e deverão ser tratados pelos blocos *catch* mais internos.

Considerando-se tais observações, serão criados pelo menos cinco casos de teste.

No primeiro caso de teste, deverá ocorrer uma exceção ao se executar o nó 297. No segundo, deverá ocorrer uma exceção no nó 371. No terceiro, uma exceção no nó 544. No quarto e quinto, uma exceção no nó 606, uma pelo caminho em que passa pelos nós 371 e 380, e outra pelo caminho que exerceite os nós 544 e 553.

Para os elementos requeridos $\langle \text{tabela_funcionario}, \langle 0,0 \rangle, (371, 414) \rangle$ e $\langle \text{tabela_funcionario}, \langle 0,0 \rangle, (544, 579) \rangle$, serão planeados dois casos de teste em que ocorra uma exceção ao se tentar alterar e outra ao inserir os dados.

No caso da alteração, tentou-se modificar um campo colocando seu valor dentro de apóstrofos, como na Figura 70. Ao ser exercitado, não foi possível realizar a alteração. Deve-se, então, observar se o resultado era o esperado. Por um lado, não houve um erro fatal ou inconsistência de dados.

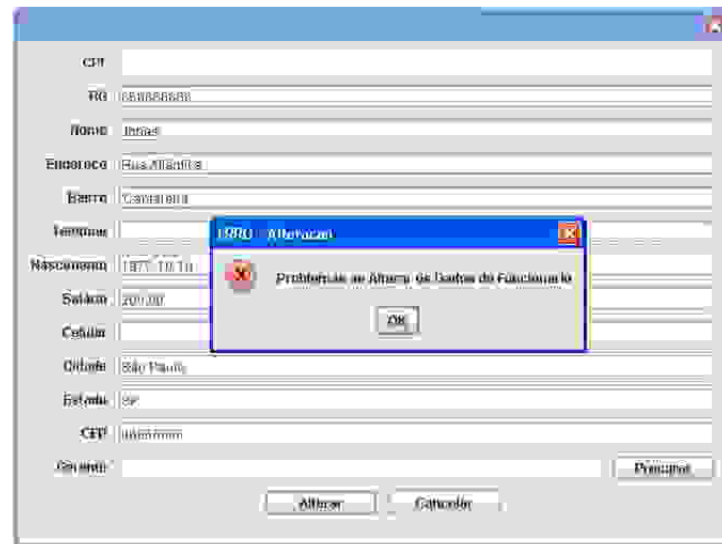


Figura 70: Alteração com Dado Inválido

O Sistema alertou o usuário de que a operação não pôde ser realizada, então, caso o documento de requisitos não exija outro comportamento, pode-se interpretar o resultado obtido como o esperado.

Para a inserção, tentou-se inserir um funcionário que possua um apóstrofo no nome, como Joana D'Arc. O resultado pode ser observado na Figura 71.

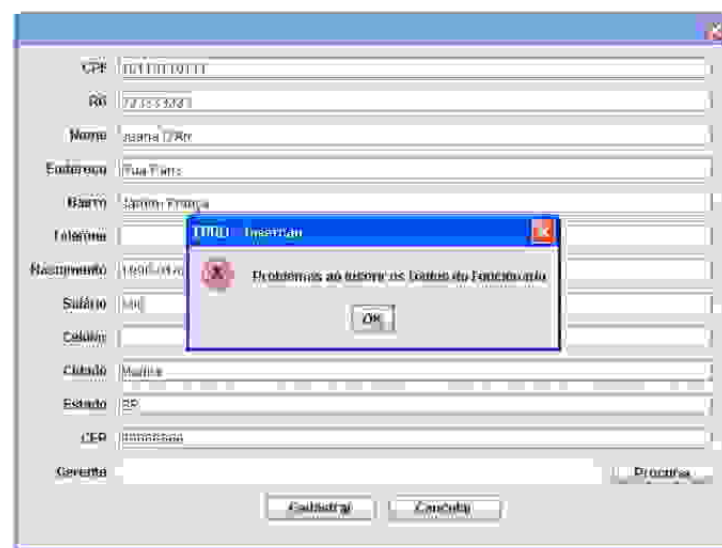


Figura 71: Inserção com Dado Inválido

Deve-se observar que existem clientes que possuem apóstrofo nos nomes. Se o Sistema não permite o uso de apóstrofo, nem trata nomes removendo-o, se presente, significa que determinados nomes não poderão ser inseridos, podendo ser considerado um erro.

Os elementos <tabela_funcionario, <544,553>,(606, 633)> e <tabela_funcionario, <371,380>,(606, 633)> dependem que uma exceção ocorra no nó 606. Porém, o comando SQL é um select simples, sem presença de join ou alguma falha que possa ser explorada, mesmo que se receba uma consulta vazia. Desta forma, ambos os elementos foram considerados inefectivos neste exemplo (ainda que se possa forçar seus exercícios derrubando-se o banco de dados logo antes de ser executada a instrução relacionada ao t-uso).

Segue, na Figura 72, o resultado da análise de cobertura do critério todos-t-usos-dependente de exceção.

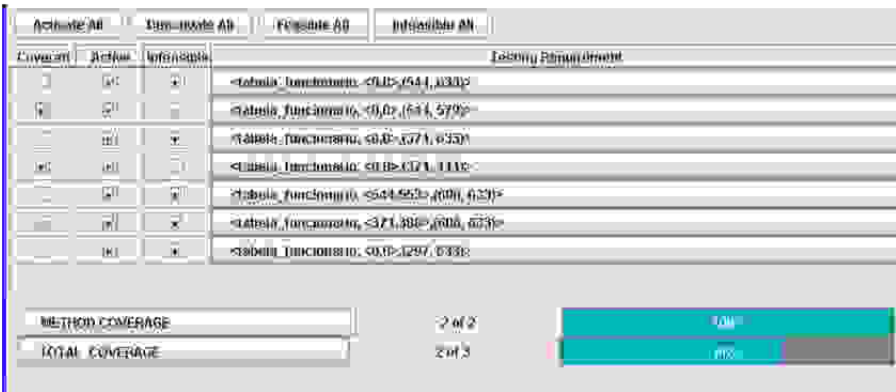


Figura 72: Resultado de Cobertura dos Elementos Requeridos

Os elementos requeridos inefectivos (*infeasible*) foram marcados pelo usuário de teste. Observa-se que a cobertura do método só levou em consideração os elementos factíveis.

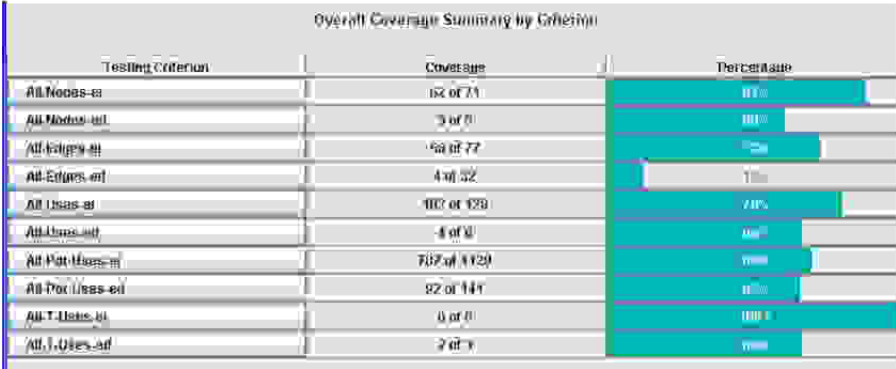


Figura 73: Sumário da Análise de Cobertura por Critério

Por fim, a JaBUTi oferece diversas formas de relatório, desde a compilação detalhada de cada dado estatístico levantado pela ferramenta, em formato HTML, a sumários como o mostrado na Figura 73.

Esta seção apresentou a aplicação da JaBUTi como ferramenta de automatização de parte das etapas envolvidas no processo de teste com o uso do critério todos-t-usos, dividido em todos-t-usos-independente de exceção e todos-t-usos-dependente de exceção,

6.2 Conclusão

Este capítulo apresentou um estudo de caso, executando-se um conjunto de casos de teste em um Sistema para lojas de cosméticos. Foi selecionado um método para ser testado, em que ocorrem as ações de cadastramento e alteração de dados de funcionário.

A primeira seção descreveu as principais características do Sistema utilizado e do método selecionado para o teste. A segunda seção abordou a aplicação da JaBUTi com objetivo de apresentar a dinâmica do teste com o uso desta ferramenta e evidenciar as vantagens de sua aplicação. Esta seção mostrou as facilidades da automatização de parte da etapa de teste.

Como exemplo das facilidades fornecidas, foi apresentada a diferenciação dos nós de comandos SQL, que permitem o rápido reconhecimento dentro de um método. Foi apresentada a capacidade da ferramenta de detectar comando SQL automaticamente, se estiver na forma de uma constante, e a possibilidade de se entrar com a expressão SQL manualmente dentro do nó de Comando SQL. Também foi mostrada a identificação dos elementos requeridos com base no fluxo de dados, a capacidade de determinar quais destes foram exercitados pelos casos de teste, a análise de cobertura fornecida e a apresentação dos dados em forma de relatórios.

7 Conclusão e Trabalhos Futuros

Foi apresentada, nesta dissertação, uma solução para a implementação do critério de teste de Aplicações de Bancos de Dados Relacionais, todos-t-usos (SPOTO, 2000), na JaBUTi. Este critério enfoca os testes de unidade e pertence a uma família de critérios que abrange testes de unidade e de integração.

A JaBUTi é uma ferramenta para o entendimento e realização de testes de unidade em bytecode, que suporta os critérios todos-nós, todos-arcos, todos-usos, todos-potenciais-usos e todos-t-usos. Dentre suas funcionalidades, cria o grafo de fluxo de dados, identifica os elementos requeridos para cada critério suportado e a cobertura de cada elemento após a execução dos casos de teste.

O projeto desenvolvido é parte de um trabalho maior, que visa ao desenvolvimento contínuo da ferramenta em questão. A proposta inicial objetivava a implementação da capacidade da JaBUTi em detectar os nós de Comando SQL, as associações entre definição e t-uso e os elementos requeridos, sem considerar os nós commit. A ferramenta também deveria ser capaz de detectar os elementos exercitados, apresentar a distribuição de pesos e os resultados da cobertura.

Na fase de estudo do código da JaBUTi e da implementação do critério todos-t-usos, foi observada a viabilidade de se ir um pouco além do proposto. Foi inserida a capacidade de se considerar o nó *commit* na detecção dos elementos requeridos e a possibilidade do usuário de teste entrar manualmente com uma expressão SQL em um nó SQL ao qual a ferramenta não tenha conseguido identificá-la.

Durante a implementação, o modelo já existente para o critério todos-usos foi utilizado como base para a codificação, tendo-se realizadas as adaptações necessárias para respeitar as diferentes características em relação ao critério todos-t-usos, como a criação da classe AllTUses, baseada em AllUses. Considerando-se que os algoritmos responsáveis pelas funcionalidades já existentes não foram alterados, não houve mudança significativa

na estrutura do código original.

Um estudo de caso foi apresentado para elucidar a aplicação da Ferramenta JaBUTi ao se utilizar o critério todos-t-usos, e apresentar as vantagens da automação de parte do processo de teste.

Uma questão levantada nas fases iniciais do projeto foi a aplicabilidade do critério para testes de unidade em programas orientados a objetos. Spoto (2000), em sua tese, demonstrou que tais critérios de unidade para aplicações de bancos de dados são aplicáveis em programas não orientados a objeto. Mas, para este paradigma, pode-se escolher desenvolver métodos que representem uma única ação. Por exemplo, em uma classe Funcionário, poderia haver um método para cadastramento, um método para inserção de dados, um para alteração e um para apagá-lo. Isso diminuiria ou anularia a existência de associações entre definição persistente e t-uso em um mesmo método, já que ocorreria apenas um comando SQL por método, ou poucos.

Porém, mesmo que tal hipótese seja verdadeira, a família de critérios ABDR possui grande aplicabilidade em testes de integração. Como citado, a ferramenta está em contínuo desenvolvimento e um dos passos seguintes será implementar critérios de testes neste escopo, sendo que o projeto realizado poderá ser aproveitado ao se inserir os critérios de ABDR de integração. Dentre as funcionalidades que podem ser aproveitadas, cita-se sua capacidade de determinar os nós SQL, de inserir expressões manualmente, de determinar as associações entre definição persistente e t-uso, a representação diferenciada de nós SQL no grafo e os dados de definição e t-uso mostrados para cada nó. Ainda, a dissertação mostrou que a automatização de grande parte das etapas da execução de testes com o critério todos-t-usos é viável.

Em projetos futuros, poderá ser implementada a capacidade da ferramenta em identificar se a tupla exercitada na definição persistente está contida no t-uso, um requisito dos critérios ABDR.

Uma posterior implementação do critério ABDR de unidade, todos-dtu-caminhos, também poderá aproveitar as mesmas funcionalidades citadas para os eventuais critérios de ABDR para testes de integração.

Referências

- AGRAWAL, H. **Dominators, Super Block and Program Coverage**. *SIGPLAN - SIGACT Symposium on Principles of Programming Languages - POPL'94*, pag 25-34. ACM Press. Portland, Oregon, janeiro, 1994.
- ANSI; ISO; IEC. **Database Language SQL - Part 2: foundation** (sql - foundation). *International Standard (IS), ISO/IEC 9075-2:1999 (E)*, 1999.
- BECARI, G. M. **A Eficácia do Teste Estrutural em Aplicação de Banco de Dados Relacional**. 2006. *Dissertação de Mestrado, Fundação de Ensino "Eurípides Soares da Rocha", Marília, Brasil*, 2006.
- BEIZER, B. **Software Testing Techniques**. 2nd ed. New York: Van Nostrand Reinhold Company, 1990.
- CHAIM, M. L. **Poke-tool: uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados**. *Dissertação de Mestrado, UNICAMP, Campinas, Brasil, abril*, 1991.
- CHAYS, D. et al. **A Framework for Testing Database Applications**. *ISSTA2000: International Symposium on Software Testing Analysis*. Portland, agosto, 2000.
- CHUSHO, T. **Test Data Selection and Quality Estimation Based on the Concept of Essencial Branches for Path Testing**. *IEEE Trans. Software Eng.* Vol.13, No. 5, pp. 509-517. Maio, 1987.
- DELAMARO, M. E. **Mutação de Interface: um critério de adequação interprocedimental para o teste de integração**. 1997. *Tese de Doutorado, USP, São Carlos, Brasil*, 1997.
- FRANKL, P. G.; WEYUKER, E. J. **An Applicable Family of Data Flow Testing Criteria**. *IEEE Transactions on Software Engineering*, V.14 N.10, outubro, 1988.
- FRANKL, P. G.; WEYUKER, E. J. **Provable Improvements on Branch Testing**. *IEEE Transactions on Software Engineering*, V.19 N.10, outubro, 1993.
- GHEZZI, C.; JAZAYERI, M.; MANDRIOLI, D. **Fundamentals of Software Engineering**. ed. Englewood Cliffs, 1991.
- GOODENOUGH, J. B.; GERHART, S. L. **Toward a Theory of Test Data Selection**. *IEEE Transactions on Software Engineering*, V. SE-1, junho, 1975.
- HAMLET, D.; TAYLOR, R. **Partition Testing Does Not Inspire Confidence**. *IEEE Transactions on Software Engineering*, V.16 N.12, dezembro, 1990.

- IEEE. **Standard glossary of software engineering terminology**. Padrão 610.12, IEEE, 1990.
- MALDONADO, J. C. **Critérios Potenciais Usos: Uma Contribuição ao Teste de Software**. 1991. *Tese de Doutorado, DCA/FEEC/UNICAMP - Campinas, SP, Brasil*, 1991.
- MALDONADO, J. C.; CHAIM, M. L.; JINO, M. **Seleção de Casos de Testes Baseada nos Critérios Potenciais Usos. II SBES, Canela, RS, Brasil, Outubro.**, 1991.
- MORELL, L. J. **A Theory of Fault-Based Testing**. *IEEE Transactions on Software Engineering*, SE-10:795-803, novembro, 1990.
- MYERS, G. **The Art of Software Testing**. Wiley, New York, 1979.
- NARDI, P. A.; DELAMARO, M. E.; VINCENZI, A. M. R. E. S. S. **JaBUTi/BD: utilização de critérios estruturais em aplicações de bancos de dados java**. 20o *Simpósio Brasileiro de Engenharia de Software - SBES, Uberlândia-MG. 12 Sessão de Ferramentas. Outubro*, 2005.
- NTAFOS, S. C. **On Required Element Testing**. *IEEE Transactions on Software Engineering*, V. SE-10, N.6, novembro, 1984.
- NTAFOS, S. C. **A Comparison of Some Structural Testing Strategies**. *IEEE Transactions on Software Engineering*, V.14, N.6, junho, 1988.
- PERRY, W. **Effective methods for software testing**. New York: John Wiley and Sons, 1995.
- PRESSMAN, R. S. **Software Engineering: a practitioner's approach**. 5. Ed. New York: McGraw-Hill, 2001.
- RAPPS, S.; WEYUKER, E. J. **Data Flow Analysis Techniques for Test Data Selection**. *International Conference on Software Engineering, Tokio, Japan, September*, 1982.
- RAPPS, S.; WEYUKER, E. J. **Selecting Software Test Data Using Data Flow Information**. *IEEE Transactions on Software Engineering*, SE-11, abril, 1985.
- SPOTO, E. S. **Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional**. 2000. *Tese de Doutorado, UNICAMP, Campinas, Brasil*, 2000.
- VINCENZI, A. M. R. **Orientação a Objetos: definição, implementação e análise de recursos de teste de validação**. 2004. *Tese de Doutorado, USP, São Carlos, Brasil*, 2004.
- VINCENZI, A. M. R. et al. **JaBUTi - Java Bytecode Understanding and Testing: user's guide**. São Carlos, Brasil, 2003.
- ZHU, H. **A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria**. *IEEE Transactions on Software Engineering*, VOL.22, NO.4, APRIL, 1996.

Apêndice A

Este apêndice apresenta o código do programa utilizado como exemplo no Capítulo 3, para ilustrar o uso dos critérios todos-nós, todos-arcos e todos-usos, através da ferramenta JaBUTi.

A primeira classe, Demo, possui um método principal que requisita três dados de entrada. Um método da classe Exemplo é chamado e calcula o seno do primeiro dado de entrada e o representa em um gráfico. Se o valor for maior que 360, o programa deve considerá-lo como 360. Os dois outros dados de entrada são utilizados para ilustrar possíveis erros que podem ser encontrados com o uso dos critérios, não havendo outra função dentro do programa. O método PreencheEspaco(), da classe Exemplo, desenha o gráfico. O método FatorZ() é utilizado apenas para mostrar a representação do fluxo de controle gerado por comandos como *if* e laços.

Programa Programa Seno

```
package Demo;
import java.io.*;
public class Demo {
    public static void main(String[] args)
    {
        int x,z;
        String LimZ;
        Teste MeuTeste = new Teste();
        BufferedInputStream bis = new BufferedInputStream(System.in);
        BufferedReader rd = new BufferedReader(new InputStreamReader(bis));
        try
        {
            System.out.println ("Digite um valor para x:");
            x=Integer.parseInt(rd.readLine());
            System.out.println ("Digite um valor para z:");
            z=Integer.parseInt(rd.readLine());
            System.out.println ("Digite o valor de LimZ (de 0 a 360 graus):");
            LimZ=rd.readLine();
            rd.close();
            MeuTeste.Exemplo(x,z,LimZ);
        }catch(Exception e)
        {

```

```

        System.out.println ("Erro Inesperado!");
    }
}

}

*****

package Demo;
import java.lang.Math.*;
public class Teste {
    int LimZ;
    public void Exemplo(int x,int z, String TextBox)
    {
        int Contador = 0;
        String Espaco=" ";
        //todos-nós para todos-arco
        if (Integer.parseInt(TextBox) > 300)
        {
            TextBox = "300";
            LimZ = Integer.parseInt(TextBox);
        }
        System.out.println("Valor de LimZ = " + LimZ);
        //todos-arco para todos-usos;
        if (x == 0)
            x = 0;
        else
            z = z - x;
        if (z > 1)
            z = z/x;
        else
            z = 0;
        //Desenha um arco senóide
        System.out.println(-1*****0*****1);
        try
        {
            while (Contador < 301)
            {
                Espaco=PreencheEspaco(Contador,LimZ);
                System.out.println(Espaco);
                Contador = Contador + 5;
            }
        }catch(Exception e)
        {
            System.out.println("Houve um erro ao desenhar o arco.");
            System.out.println("Valores pertinentes:");
            System.out.println("Contador="+ Contador);
            System.out.println("LimZ="+ LimZ);
            System.out.println("Espaco="+ Espaco);
        }
        System.out.println("FIM");
    }
    private String PreencheEspaco(int Contador,int LimZ) throws Exception

```

```

{
    String X =
    int DesLimZ=(int)Math rint((Math.sin(Math.toRadians(LimZ))*100/3)+40);
    int DesSenq=(int)Math rint((Math.sin(Math.toRadians(Centador))*100/3)+40);
    for (int Roda = 1; Roda <= 75; Roda++)
        if (Roda == DesLimZ)
            X = X+" ";
        else
            if (Roda == DesSenq)
                X = X+"00";
            else
                if (Roda == 40)
                    X = X+"* ";
                else
                    X = X+";

    return X;
}
private void FatorZ()
{
    int z = 1;
    //IF
    if(z < 10)
        System.out.println("IF");
    //IF-ELSE
    if(z < 50)
        System.out.println("IF DO IF-ELSE");
    else
        System.out.println("ELSE DO IF-ELSE");
    //CASE
    switch (z)
    {
        case 1:
            System.out.println("z=1!");
            break;
        case 2:
            System.out.println("z=9!");
            break;
        case 3:
            System.out.println("z=3!");
            break;
        default:
            System.out.println("z < 1 ou z > 3!");
    }
    //FOR
    for (int Roda = 1; Roda < 3; Roda++)
        System.out.println("Isto é um FOR");
    //WHILE
    while (z < 10)
    {
        System.out.println("Isto é um while");
        z = z + 1;
    }
    //DO-WHILE
    do

```

```
    {  
        System.out.println("Isto é um DO");  
        if (x == 3)  
            System.out.println("Z é igual a 3");  
        else  
            System.out.println("Z é diferente de 3");  
        x = x + 1;  
    }while(x < 5);  
}
```

Apêndice B

Este apêndice apresenta o método de classe utilizado para a realização do estudo de caso (Capítulo 6). O método realiza o cadastramento e alteração de dados de funcionários, do Sistema de Cosméticos, aplicação utilizada em Becari(2005).

Programa Método de Cadastrar e Alterar Funcionário

```
private void evento_onClick_jButtononCadastrar() {
    Banco bd = new Banco();
    bd.conectaBanco();
    //VERIFICA CAMPOS
    String cpf = jTextFieldCPF.getText();
    String rg = jTextFieldRG.getText();
    String nome = jTextFieldNome.getText();
    String endereco = jTextFieldEndereco.getText();
    String bairro = jTextFieldBairro.getText();
    String telefone = jTextFieldTelefone.getText();
    String nascimento = Operacoes.converteData(jTextFieldNascimento.getText());
    String salario = jTextFieldSalario.getText();
    String celular = jTextFieldCelular.getText();
    String cidade = jTextFieldCidade.getText();
    String estado = jTextFieldEstado.getText();
    String cep = jTextFieldCEP.getText();
    String cpf_gerente = jTextFieldCPFGerente.getText();
    boolean VerificaCPF = verificaVazio(cpf,"CPF");
    boolean VerificaRG = verificaVazio(rg,"RG");
    boolean VerificaNome = verificaVazio(nome,"Nome");
    boolean VerificaEnd = verificaVazio(endereco,"Endereco");
    boolean VerificaBairro = verificaVazio(bairro,"Bairro");
    boolean VerificaData = verificaVazio(nascimento,"Data de Nascimento");
    boolean VerificaSalario = verificaVazio(salario,"Salário");
    boolean Verifica=VerificaRes(VerificaCPF,VerificaRG, VerificaNome,VerificaEnd,VerificaBairro, VerificaData,Veri-
ficaSalario);
    try{
        Statement statement = bd.getConnection().createStatement();
        VarGlobais.statement=bd.getConnection().createStatement();
        switch (parametro){
            case Estados.CADASTRAR:{
                VarGlobais.cpf_funcionario = jTextFieldCPF.getText();
                boolean verCPF=VerificaCPF(VarGlobais.cpf_funcionario);
                if (!verCPF)
                    return;
            }
        }
    }
}
```

```

//verificar se existe um funcionario já cadastrado com esse cpf
query = QueryBD.select116(VarGlobals.cpf_funcionario);
String
/*****
*SELECT*
*****/
ResultSet resultado = statement.executeQuery(query);
boolean moreRecords = resultado.next();
if (!moreRecords){
    nascimento = Operacoes.converteData(nascimento);
    if (Verifica){
        query = QueryBD.select117(cpf, rg, nome, endereco, bairro, telefone, nasci-
mento, salario, celular, cidade, estado, cep, cpf_gerente);
        try{
            /*****
            *INSERT*
            *****/
            statement.executeUpdate(query);
            bd.connection.commit();
            JOptionPane.showMessageDialog(null, "Dados do Funcionario Inseri-
dos.", "Insercao", JOptionPane.INFORMATION_MESSAGE);
            fechaJanela();
        } catch (Exception e){
            bd.connection.rollback();
            TesteBD.pontaProvaSTR(, "438 ");
            JOptionPane.showMessageDialog(null, "Problemas ao Inserir os "+
"Dados do Funcionario", "ERRO - Insercao", JOptionPane.ERROR_MESSAGE);
            return;
        }
    } else {
        return;}
} else {
    JOptionPane.showMessageDialog(null, "CPF: "+ VarGlobals.cpf_funcionario + "já
cadastrado", "Erro de integridade", JOptionPane.ERROR_MESSAGE);
    return;}
} //FIM
if (parametro == CADASTRAR)
    break;
case Estados.ALTERAR: {
    String cpfa = jTextFieldCPF.getText();
    boolean verCPF=VerificaCPF(cpfa);
    if (!verCPF)
        return;
    nascimento = Operacoes.converteData(nascimento);
    if (Verifica){
        String query = QueryBD.select132(cpf, rg, nome, endereco, bairro, telefone, nascimento,
salario, celular, cidade, estado, cep, cpf_gerente);
        /*****
        *UPDATE*
        *****/
        try{
            statement.executeUpdate(query);
            bd.connection.commit();
            JOptionPane.showMessageDialog(null, "Dados do Funcionario Alterados.", "Alteracao",
JOptionPane.INFORMATION_MESSAGE);

```

```

        fechaJanela();
    } catch (Exception e) {
        bd.connection.rollback();
        JOptionPane.showMessageDialog(null, "Problemas ao Alterar os "+ "Dados
do Funcionario", "ERRO - Alteracao", JOptionPane.ERROR_MESSAGE);
    }
    } else {return;}
}; //FIM
if (parametro == ALTERAR)
    break;
case Estados, VISUALIZAR: {
    }
}
}
/*****
*SELECT*
*****/
String query="SELECT tabfuncionario.cpf_funcionario as CPF, "+
    "tabfuncionario.cpf_gerente_funcionario as Gerente, "+
    "tabfuncionario.nome_funcionario as Nome, "+
    "tabfuncionario.endereco_funcionario as Endereco, "+
    "tabfuncionario.bairro_funcionario as Bairro, "+
    "tabfuncionario.cidade_funcionario as Cidade" +
    "FROM tabela_funcionario tabfuncionario";
VarGlobais.ResFuncionario=VarGlobais.statement.executeQuery(query);
statement.close();
} catch (Exception e){
    System.out.println("Não foi possível realizar a tarefa ->" + e);
}
}

```