FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” - UNIVEM
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GISELE MOLINA BECARI

A EFICÁCIA DE CRITÉRIOS DE TESTE ESTRUTURAL EM APLICAÇÃO DE BANCO DE DADOS RELACIONAL

Marília - SP
2005

GISELE MOLINA BECARI

A EFICÁCIA DO TESTE ESTRUTURAL EM APLICAÇÃO DE BANCO DE DADOS RELACIONAL

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípedes de Marília, mantido pela Fundação de Ensino Eurípedes Soares da Rocha, para obtenção do Título de Mestre em Ciência da Computação (Área de concentração: Engenharia de Software.)

Orientador:
Prof` Dr Edmundo Sérgio Spoto

Marília - SP
2005

BECARI, Gisele Molina
A EFICÁCIA DE CRITÉRIOS DE TESTE ESTRUTURAL EM APLICAÇÃO DE BANCO DE DADOS RELACIONAL / Gisele Molina Becari; orientador: Edmundo Sergio Spoto. Marília, SP: [s.n.], 2005.
106 f.

Dissertação (Mestrado em Ciência da Computação) – Centro Universitário de Marília – Fundação de Ensino Eurípedes Soares da Rocha.

CDD:

GISELE MOLINA BECARI

A EFICÁCIA DE CRITÉRIOS DE TESTE ESTRUTURAL EM APLICAÇÃO DE BANCO DE DADOS RELACIONAL

Banca Examinadora da dissertação apresentada ao programa de Mestrado da UNIVEM. / F.E.E.S.R., para obtenção de Título de Mestre em Ciência da Computação. Área de concentração: Engenharia de Software.

Resultado: _____

ORIENTADOR PROF DR : EDMUNDO SÉRGIO SPOTO

1º. EXAMINADOR: _____

2º. EXAMINADOR: _____

MARÍLIA, 24 DE OUTUBRO DE 2005

Dedico este trabalho a minha amada família que Deus colocou como uma dádiva da minha vida: meu marido Márcio André Becari que eu amo muito e sempre está do meu lado, meus pais Maria Zilda Fugolin Molina, Rubens Molina (in memorian) que lutaram muito e venceram, minhas irmãs guerreiras Ana Heloisa Molina, Fátima A. Molina Sanches, minhas amigas a qualquer hora, meu irmão cunhado Adalberto Sanches Munaro, dedicado e companheiro e minhas sobrinhas fofas Isadora Molina Sanches e Raquel Molina Sanches, as alegrias de todos nós.Em qualquer situação eu sei que sempre posso contar com estas pessoas Maravilhosas.

AGRADECIMENTOS

Pessoas muito especiais estiveram comigo na geração desta dissertação, de uma forma ou outra contribuíram com seu apoio, entusiasmo, dicas, estudos, paciência, um abraço.

Gostaria de agradecer a Deus, pela oportunidade de estar viva, com saúde e permitir realizar este trabalho.

Agradeço meu orientador Prof. Dr.Edmundo Sérgio Spoto, pela paciência, orientação e confiança que me passou em todas reuniões, e sempre corrigindo o curso do trabalho, quando necessário, colocando meus pés no chão. A luta foi intensa. Obrigada por tudo.

A todos os professores que ministraram as aulas deste curso, acrescentando todo conhecimento necessário: Prof. Dr.Jorge Silva, Prof. Dr. Marcos Mucheroni, Prof. Dr. Marcio Eduardo Delamaro e Prof. Dr. Plínio Vilela.

Agradeço também a Unilins, com a ajuda de custo de viagem e o Coordenador do Curso Wagner Dizeró.

À turma da classe, que foi um verdadeiro presente conhecer tanta gente amiga, lembrando os momentos de churrasco, almoço na “padóca”, ou almoço e passeio no shopping e também o desespero das provas, trabalhos, entrega da qualificação e finalmente a entrega do trabalho final: Claudete Werner, minha querida amiga de todas as horas, Afonso, Andréia Machado, Ângela Rochetti, Lucia Shiraisi, Fábio Montanha, César Cusin, Fabião, Luis Fernando (Segal), Júnior, Eduardo Damaceno (Travaceno), André (Filhão), Ricardo Veronesi (RV), Marcio Cardim (Perpétua), André Gobbi, Wagner (Wagnão), Fernando Riquelme (Smurf), Leila Haga, Wendel Brusolini, Christiano, José Merlin, Alexandre, Marçal, Fabinho, Cesinha, Marcelo (Bariri), Cássio, Luciana (Lule), Thiago e Sérgio. Aos novos amigos que colaboraram com o Sistema de Cosméticos deste trabalho: Gustavo Garcia Rondina, Rodrigo Fraxino de Araújo e Marcelo Rossi. Outros amigos que conheci no decorrer do curso e aos poucos fizeram parte da turma que de alguma forma apoiaram este trabalho: Paulo Augusto Nardi, Larissa Pavarin, Paulinha, Vasco, Gustavo Herberman, Mario, Leninha e Beth.

Aos amigos e parentes que sempre que sempre torceram por mim: Tio Julinho e Tia Zilda, Tia Irma, André e Andréa, Patrícia Helena, Marcinha, Denise, Patrícia Rigoto e Maria Luiza e Eduardo.

À minha família querida: meu marido Márcio André Becari, minha mãezinha Maria Zilda Fugolin Molina, meu pai Rubens Molina que já está no “andar de cima”, minhas “Ermãs” Ana Heloisa Molina e Fátima Ap. Molina Sanches, juntamente com Adalberto Sanches Munaro (Binha: meu irmão) e minhas sobrinhas fofas Isadora Molina Sanches e Raquel Molina Sanches.

"... Queira
Basta ser sincero e desejar profundo
Você será capaz de sacudir o mundo, vai
Tente outra vez..."
"...Tente
E não diga que a vitória está perdida
Se é de batalhas que se vive a vida
Tente outra vez..."
(**Raul Seixas**)

"...Penso que cumprir a vida seja simplesmente
compreender a marcha, e ir tocando em frente
Cada um de nós compõe a sua história,
e cada ser em si, carrega o dom de ser capaz,
e ser feliz..."
(**Almir Sater**)

BECARI, Gisele Molina. A Eficácia de Critérios de Teste Estrutural em Aplicação de Banco de Dados Relacional. 2005. 106 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino Eurípedes Soares da Rocha, Marília. 2005.

RESUMO

O teste em Aplicação de Banco de Dados Relacional distingue-se do teste em aplicação convencional pela manipulação de dados persistentes, na qual esses dados não são voláteis na execução da aplicação. Estudando as várias estratégias e técnicas estruturais de teste, abrangendo seus critérios, relacionados à Aplicação de Banco de Dados Relacional (ABDR), verificou-se a força que os critérios têm em relação à detecção do erro comparando-os com outros critérios que não detectam tais erros. Um dado critério de teste exercita um conjunto de possíveis ocorrências de erros associados a um conjunto de características de restrições de Banco de Dados Relacional. Esses critérios são divididos em fluxos de dados Intra-Modular (Intra-Classe) e Inter-Modular (Inter-Classe), tendo em vista que a ABDR do estudo de caso está implementada em Linguagem Java, que exercita diferentes associações, definição e uso persistente entre diferentes *variáveis tabela* de uma Base de Dados Relacional. Foram propostas algumas sugestões de melhoria da escolha dos dados de teste para torná-lo mais eficaz na detecção de defeitos, com base nas restrições existentes no modelo de banco de dados relacional. Foi apresentado um estudo de caso, bem como as análises dos resultados obtidos com base nos critérios ABDR de Spoto (2000) (fluxo de dados Intra-Modular e Inter-Modular). Para finalizar, são apresentadas conclusões e sugestões para trabalhos futuros.

Palavras Chaves: Teste Estrutural em Aplicação de Banco de Dados Relacional, SQL, Critério de Teste, Eficácia do Teste, Tipo de defeito.

BECARI, Gisele Molina. A Eficácia de Critérios de Teste Estrutural em Aplicação de Banco de Dados Relacional. 2005. 106 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino Eurípedes Soares da Rocha, Marília. 2005.

ABSTRACT

Testing in Relational Database Applications distinguishes from testing in conventional application by the manipulation of persistent data, in which these data are not volatile in the execution of the application. This work show that dataflow testing criterion exercise a set of possible occurrences of errors associated to a set of characteristics of restrictions of Database Relational studying the some strategies and structural testing techniques, enclosing its criteria, related to the Relational Database Applications (RDA) it is verify the force that the criteria have in relation to the detection of the error and to compare them with other criteria that do not detect such errors. These criteria are divided in flows of data Intra-Modular (Intra-Class) and Inter-Modular (Inter-Class), in view of that the RDA of study of case is implemented in Java Language, that different associations exercise definition and persistent use between different *variable prices* of a Database Relational. Some suggestions of improvement of the choice of the data of test to become it more efficient in the detention of defects had been proposals, on the basis of the existing restrictions in the model of database relational. A case study was presented, as well as the analyses of the gotten results on the basis of criteria RDA of Spoto (2000). At the end, the conclusions and suggestions of future works are presented.

Keywords: Structural testing in Relational Database Applications, SQL, Criteria of Testing, Effectiveness of the Test, Type of defect.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABDR – Aplicação de Banco de Dados Relacional

CRA – Com respeito a

DCL - Linguagem de Especificação de Restrições

DDL - Linguagem de Definição de Dados

DER – Diagrama de Entidade e Relacionamento

DML - Linguagem de Manipulação de Dados

RIs - Restrições de Integridade

SGBD - Sistema Gerenciador de Banco de Dados

SQL - *Structured Query Language*

FCPU - Família de Critérios Potenciais Usos

FCPD - Família de Critérios de Fluxos de Dados

SUMÁRIO

1. INTRODUÇÃO

Nos últimos vinte anos a Engenharia de Software vem contribuindo com novas técnicas para obtenção de qualidade dos softwares produzidos e disponibilizados no mercado. O teste de software é uma das fases do ciclo de vida utilizada na melhoria da qualidade. Quando introduzida no processo de desenvolvimento é um dos requisitos da construção do software.

Segundo Sommerville (2003), um teste bem sucedido para detecção de defeitos é aquele que faz com que o sistema opere incorretamente e, como conseqüência, expõe um defeito existente. Isto enfatiza a importância sobre os testes e demonstra a presença, e não a ausência, de defeitos de programa. Os casos de teste são especificações das entradas para o teste e da saída esperada do sistema, mais uma declaração do que está sendo testado. Os dados de teste são as entradas que foram criadas para testar o sistema.

Em Maldonado et. al. (1991) tem-se verificado que embora se gaste em geral 50% do orçamento para desenvolvimento do software em atividades de teste, um número significativo de defeitos permanecem sem serem detectados nos softwares testados e liberados. Esses defeitos normalmente têm um impacto grande na operação normal do sistema.

Teste é uma forma de verificar se o que está sendo desenvolvido está de maneira correta conforme os requisitos especificados pelo usuário. Os testes são essenciais para o controle de qualquer projeto de desenvolvimento de software. O teste de software envolve: planejamento de testes, projeto de casos de testes, execução e avaliação dos resultados obtidos. É uma forma de medir a qualidade do software (PRESSMAN, 1995).

O teste consiste basicamente em executar um programa fornecendo dados de entrada e comparar a saída alcançada com o resultado esperado, obtido na especificação do programa (MYERS 1979).

Alguns fatores básicos são necessários para comparar os critérios de teste, como: o custo (esforço necessário para que o critério seja usado), a eficácia (capacidade que um critério possui em detectar um maior número de erros em relação a outro critério) e a dificuldade de satisfação (probabilidade de satisfazer um critério tendo satisfeito outro) (MALDONADO et al,1991).

Na tentativa de reduzir custos, são propostas várias técnicas e vários critérios que auxiliam na condução e avaliação das atividades de teste. A diferença entre essas técnicas está basicamente na origem da informação que é utilizada para avaliar ou construir conjuntos de casos de teste, sendo que cada técnica possui uma variedade de critérios para esse fim (VINCENZI, 1998).

A eficácia de um critério de teste está relacionada à habilidade do critério em levar o testador a selecionar dados que tenham uma boa chance de revelar os defeitos do programa (BATISTA, 2003).

Uma das principais atividades em teste de software é o projeto e avaliação de casos de teste. Atualmente, utiliza-se um conjunto de técnicas, métodos e critérios. As técnicas podem ser classificadas basicamente em: funcional, estrutural, baseada em erros e baseada em estado (ou uma combinação destas), dependendo da origem da informação que é utilizada para derivar os requisitos de teste (elementos requeridos).

Segundo Vincenzi et al. (2003), as técnicas e critérios de teste constituem um mecanismo para avaliar a qualidade da atividade de teste.

Estas considerações são apoiadas por Freedman (1991), que apontam para as pesquisas desenvolvidas na área de teste que se concentram em dois problemas:

a) Eficácia do teste (*Test Effectiveness*): Qual a melhor forma de selecionar dados de teste, de modo a maximizar o número de defeitos revelador?

b) Adequação do Teste (*Test Adequacy*): Como saber se os testes realizados são suficientes?

Sabendo-se que em geral o teste exaustivo (executar um programa com todo o seu domínio de entrada), é impraticável, a eficácia do teste está relacionada à geração do menor conjunto de dados de entrada para os quais as saídas produzidas irão resultar na descoberta do maior número de defeitos possível (VINCENZI et al, 2003). Conforme pode ser visualizado na Figura 1.1, onde E1, E2, E3 e E4 são os dados de entrada, P é o programa e D1, D2...D6, são os defeitos (em maior número).

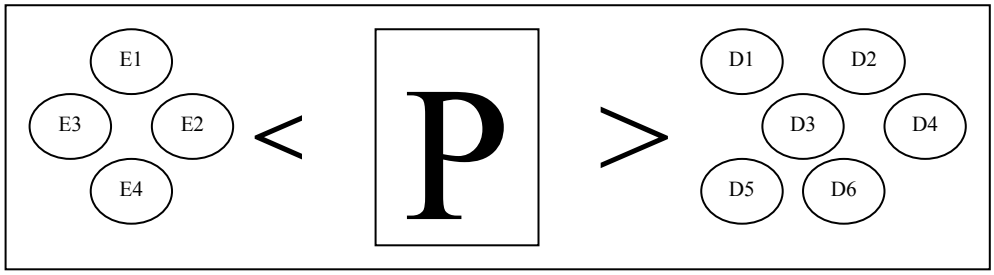


Figura 1.1: Eficácia: menos dados de entrada, revelando maior número de defeitos

Refletindo o termo "defeito", Vilela (1998) o relaciona em conjunto com "erro" e "falha" com os termos em inglês: "*fault*", "*error*", "*failure*", respectivamente, onde seus significados são: i) **Defeito**: é uma deficiência mecânica ou algorítmica que, se ativada, pode levar a uma falha; diz-se que o defeito é inerente ao programa e é geralmente inserido por um "engano" ("*mistake*") do programador ou do analista, ii) **Erro**: é um item de informação ou estado inconsistente do programa; considerado como um item dinâmico que geralmente surge após a ativação de um defeito; iii) **Falha**: é o evento notável no qual o programa viola suas especificações. Uma condição indispensável para a ocorrência de uma falha é que, pelo menos um defeito seja ativado (neste caso dize-se que a execução exercitou um ou mais defeitos).

Pressman (1995) afirma que uma estratégia de teste de software deve ser flexível o bastante para promover a criatividade e a customização necessárias para testar adequadamente todos os grandes sistemas baseados em software. Ao mesmo tempo, a estratégia deve ser

rígida o suficiente para promover um razoável planejamento e rastreamento administrativo à medida que o projeto progride.

O engenheiro cria uma série de casos de teste que têm a intenção de "demolir" o software que ele construiu (PRESSMAN, 1995). Na realidade, o objetivo dos testes é dar maior confiabilidade ao sistema e minimizar a possibilidade de ocorrências de falhas graves.

Hoje em dia uma das maiores riquezas de uma empresa é a informação. Essa informação, em empresas informatizadas, geralmente está armazenada em banco de dados. Portanto, é de extrema importância o uso de técnicas específicas para testar as bases de dados, principalmente antes de serem instanciadas, pois, os bancos de dados, são responsáveis por quase 80% das informações que circulam nas grandes empresas mundiais, armazenando, disponibilizando e protegendo uma numerosa quantidade de dados. Além da consistência e integridade destes, é essencial exercitar as necessidades especificadas da base de dados, de maneira tal que a empresa possa tomar decisões corretas apoiadas em informações geradas com base no banco de dados.

Os testes de bases de dados são realizados, basicamente, sobre relações e relacionamentos. Assim, os atributos do esquema de uma base de dados relacional são os elementos a serem exercitados (ARANHA et al., 2000).

Conforme Chays et al (2000), é essencial que o sistema de banco de dados funcione corretamente e forneça um desempenho aceitável. Um significativo esforço tem sido dedicado para assegurar que os algoritmos e as estruturas dos dados usados pelos Sistemas de Gerenciamento de Banco de Dados (SGBDs) trabalhem eficientemente e protejam a integridade dos dados. Porém, uma atenção relativamente pequena tem sido dada para o desenvolvimento de técnicas sistemáticas para programas de aplicação de banco de dados. Dados os papéis críticos destes sistemas em uma sociedade moderna, há claramente uma necessidade para novas abordagens para avaliar a qualidade dos programas de aplicação de

banco de dados. Nos últimos anos, as grandes empresas investiram maciçamente em elaboração de novas técnicas de desenvolvimento de software, no entanto, são escassos os esforços e as iniciativas que tratam do teste de programas de aplicação de banco de dados. Chays et al (2003) apresentaram uma ferramenta que automatiza a abordagem apresentada em (CHAYS et al, 2000), baseada na indicação do testador de dados típicos para cada atributo da base de dados. Esta ferramenta auxilia a aplicação de casos de teste e a checagem para conferir se as saídas e o novo estado da base de dados estão consistentes com o comportamento esperado.

Segundo Spoto (2005), a geração de bases de dados é uma área pertinente ao teste de aplicações que manipulam dados persistentes. A utilização de bases dedicadas à atividade de teste geralmente alcança uma reduzida amostra de dados, a qual pode não ser representativa dos cenários reais de uso, especialmente expor dados protegidos contra acessos não autorizados. Wu et. al. (2003) investigam técnicas para a aplicação de bases de reprodução ao teste de aplicações, sem revelar qualquer informação confidencial nessas bases, onde o objetivo é simular um ambiente real para propósito de teste, mantendo a privacidade dos dados na base original.

De acordo com Spoto (2005), uma estratégia de teste de software pode consistir na aplicação sistemática de diferentes critérios, de natureza distinta, e que revelam classes diversas de defeitos no programa. No processo de teste de um sistema caracterizam-se três níveis de teste: o teste de unidade, o teste de integração e o teste de sistema. O teste de unidade tem como função detectar defeitos na menor unidade do software, representada como a Unidade do Programa *UP*. O teste de integração objetiva testar as relações e interfaces entre as *UPs*, sendo conduzido após o teste de unidade para todas *UPs* do mesmo Módulo de Programa envolvido na integração. O teste de sistema, realizado após o teste de integração,

visa a identificar características de desempenho. A grande maioria dos critérios estruturais de teste é aplicada na fase do teste de unidade (MALDONADO, 1991).

Em Leitão et. al. (2005), é estudado como falhas podem ser manifestadas e construído um mapeamento entre tipo de defeitos e dimensões de falhas, visando a abstrair os fatores que influenciam a propagação de defeitos em comandos SQL, que manipulam dados persistentes, até a saída de execução de comandos.

1.1 Objetivo

O principal intuito deste trabalho é estudar os vários critérios de teste de software estrutural, relacionados a ABDR, na qual são propostos e elaborados, porém poucos deles são avaliados com visões dos tipos de erros que podem ser encontrados. Através dos critérios de teste estrutural de Aplicações de Banco de Dados Relacional, definido em Spoto (2000), utilizando uma aplicação real, pretende-se mostrar que um dado critério de teste exercite um conjunto de possíveis ocorrências de erros associado a um conjunto de características de restrições de Banco de Dados Relacional. Esses critérios são divididos em fluxos de dados Intra-Modular (em um mesmo programa) e Inter-Modular (em programas diferentes) que exercitam diferentes associações, definição e uso persistente entre as *variáveis tabela* envolvidas na Base de Dados Relacional. Para os critérios definidos por Spoto (2000), são abrangidas outras características não inseridas originalmente no teste de Aplicação de Dados Relacional (ABDR), mas que podem contribuir com outros tipos de detecção de defeitos como integridade referencial, domínio de validade de atributos, dependência de atributos da mesma tabela, dependência de atributos de tabelas diferentes, atributos não nulos, atributos únicos, entre outros, suprindo desta forma outras necessidades, complementando assim esses critérios. A partir desta seção os fluxos de dados serão tratados como Intra-Classe e Inter-Classe por utilizar-se Aplicações em Linguagem Java, sendo assim as Unidades de Programas

(UPs) podem ser interpretadas como Métodos e os Módulos de Programa como Classes quando tratar de programa com paradigma Orientado a Objeto (OO).

Estudando as várias estratégias e técnicas estruturais de teste, abrangendo critérios relacionados à Aplicação de Banco de Dados Relacional pretende-se verificar a força que os critérios têm em relação à detecção de defeitos e bem como compará-los com outros critérios em relação à cobertura dos elementos requeridos. A Ferramenta JaBUTi, foi utilizada para criação dos grafos para os critérios de Spoto, além de utilizá-la para comparação dos demais critérios já inseridos na ferramenta (todos-os-nós, todos-os-arcos, todos-os-usos, todos-os-potenciais-usos)

1.2. Motivação e Organização do Trabalho

Conforme Chays et al (2003), são escassos os esforços e as iniciativas que tratam de teste de programas de aplicação de banco de dados. Porém, a demanda de utilização de aplicação de banco de dados relacional em pequenas e grandes empresas está aumentando, e em muitos casos sem uma técnica que tratam o teste em ABDR, motivando assim este trabalho.

Foi constituído um plano de estratégias de teste no qual o testador observou quais tipos de defeitos o critério consegue detectar. Este plano auxilia a verificação do caminho percorrido na execução do caso de teste, mostrando quais tipos de comandos DML (Linguagem de Manipulação de Dados) foram executados e quais variáveis tabela foram exercitados neste percurso.

A eficácia do teste é uma forma de mostrar o quanto que o critério pode contribuir para a detecção de defeitos e em que tipos de situações isso é possível ocorrer. Outra característica deste trabalho é observar se um determinado critério é mais abrangente que outro (no sentido de suprir as necessidades de outros critérios) ou se apenas complementa um outro critério.

A organização deste trabalho segue da seguinte forma: no Capítulo 2 são apresentados terminologias e conceitos básicos utilizados em Banco de Dados Relacional. No Capítulo 3, um estudo mais detalhado de Teste Estrutural em Aplicações de Banco de Dados Relacional (unidade e integração), a eficácia do teste na detecção de erros e a Ferramenta JaBUTi. No Capítulo 4, tem-se o estudo de caso e a etapa do projeto, juntamente com o plano de teste de execução dos casos de teste e resultados obtidos dos testes realizados na Ferramenta JaBUTi. O Capítulo 5 apresenta as conclusões em relação aos levantamentos realizados. No Capítulo 6 são apresentadas as referências bibliográficas, finalizando com os Apêndices.

2. TESTE DE APLICAÇÃO DE BANCO DE DADOS RELACIONAL: TERMINOLOGIA E DEFINIÇÕES

Neste capítulo, são abordadas as definições básicas de Banco de Dados Relacional e as definições e terminologia de Teste Estrutural de Aplicações com Banco de Dados Relacional, abordando ainda os critérios de teste Estrutural de Aplicações de Banco de Dados Relacional, definidos em Spoto (2000). Como ferramenta de apoio, para geração dos grafos de programa, foi utilizada a JaBUTi – *Java Bytecode Understanding and Testing*, sendo esta uma ferramenta de teste baseada em fluxo de dados e fluxo de controle para programas e componentes Java (VINCENZI, 2003). Para este trabalho, foi utilizada a Ferramenta JaBUTi para geração dos grafos de programas de ABDR descritos em Linguagem Java com SQL (NARDI et.al., 2005).

2. Banco de Dados Relacional – definições básicas

O teste de aplicação de banco de dados relacional requer o conhecimento de alguns conceitos básicos de banco de dados relacional, bem como dos aspectos semânticos de um sistema gerenciador de banco de dados relacional (SGBDR). Sendo assim, serão apresentados a seguir conceitos e terminologias sobre banco de dados relacional.

2.1.1 Banco de Dados Relacional

Em Korth (1999), é definido que um **banco de dados relacional** consiste em uma coleção de *tabelas*, cada uma das quais com um nome único. Uma linha em uma tabela representa um *relacionamento* entre um conjunto de valores. Uma vez que essa tabela é uma coleção de tais relacionamentos, há uma estreita correspondência entre o conceito de *tabela* e o conceito matemático de *relação*, a partir das quais se origina o nome modelo de dados.

O Sistema de Gerenciamento de Banco de Dados (SGBD), segundo (DATE, 2000), é o software que trata todo o acesso ao banco de dados. Conceitualmente o que ocorre segundo este autor é o seguinte:

1. Um usuário faz um pedido de acesso usando uma determinada sublinguagem de dados (em geral, SQL).

2. O SGBD intercepta o pedido e o analisa.

3. O SGBD inspeciona, por sua vez, o esquema externo para esse usuário, o mapeamento externo/conceitual correspondente, o esquema conceitual, o mapeamento conceitual/interno e a definição da estrutura de armazenamento.

4. O SGBD executa as operações necessárias sobre o banco de dados armazenado.

Korth (1999), define SGBD como uma coleção de dados inter-relacionados e uma coleção de programas para acesso a estes dados. A meta básica de um SGBD segundo o autor, é proporcionar um ambiente conveniente e eficiente para recuperação e armazenamento de informações.

Em (KORTH, 1999) tem-se que um sistema de banco de dados é projetado para armazenar grandes volumes de informações. O gerenciamento de informações implica a definição das estruturas de armazenamento destas informações e o fornecimento de mecanismos para sua manipulação. Além disso, o sistema de banco de dados precisa proporcionar segurança ao armazenamento de informações, diante de falhas do sistema ou acesso não autorizado. O objetivo principal de um sistema de banco de dados, segundo Korth (1999), é proporcionar aos usuários uma visão abstrata dos dados. Isto é, o sistema esconde determinados detalhes de como os dados são mantidos e como estão armazenados.

Elmasri e Navathe (2005) descrevem as seguintes fases do projeto de banco de dados:

a) *coleta e análise de requisitos*: os projetistas fazem várias reuniões com os usuários do Banco de Dados para entender e documentar seus requisitos de dados. Todos os requisitos dos usuários são escritos e posteriormente especificados e detalhados;

b) *projeto de banco de dados conceitual*: com os requisitos coletados e analisados, cria-se um esquema conceitual para a base de dados usando o modelo de dados conceitual de alto nível. O esquema conceitual é uma descrição concisa dos requisitos de dados dos usuários e descrições detalhadas dos tipos de dados, relacionamentos e restrições (utilizando os conceitos fornecidos pelo modelo de dados de alto nível);

c) *projeto de banco de dados lógico ou mapeamento do modelo de dados*: o projeto de banco de dados é a implementação da base de dados usando um SGBD comercial. Seus resultados são os esquemas da base de dados a serem usados na implementação do modelo de dados do SGBD;

d) *projeto de banco de dados físico*: as estruturas de armazenamento internas e organizações de arquivos são especificadas.

2.1.2. Modelo Relacional de Dados

Segundo Date (2000), os sistemas relacionais são baseados em uma fundamentação formal, chamada **modelo relacional de dados**. O que a expressão significa segundo o autor é que, em um sistema desse tipo:

1. **Aspecto estrutural**: os dados no banco de dados são percebidos pelo usuário como tabelas, e nada além de tabelas.
2. **Aspecto de integridade**: essas tabelas satisfazem a certas restrições de integridade.
3. **Aspecto manipulativo**: os operadores disponíveis para que o usuário possa manipular essas tabelas - por exemplo, para propósitos de busca de dados – são operadores que derivam tabelas de outras tabelas.

Em (ELMASRI e NAVATHE) o modelo relacional representa o banco de dados como uma coleção de relações. Cada relação se parece com uma tabela de valores. Quando uma relação é pensada como uma tabela de valores, cada linha na tabela representa uma coleção de valores de dados relacionais. No modelo relacional, segundo os autores, cada linha na tabela representa um fato que corresponde a uma entidade ou relacionamento no mundo real. O nome da tabela e os nomes das colunas são usados para ajudar na interpretação do significado dos valores em cada linha.

Tomando uma relação como uma tabela, Date (2000) considera que uma **tupla** corresponde a uma linha desta tabela e um **atributo** corresponde a uma coluna. O número de tuplas é chamado cardinalidade e o número de atributos é chamado grau; um domínio é um conjunto de valores do qual são tomados os valores de atributos específicos de determinadas relações.

Como exemplo de Korth (1999), considerando a tabela conta da Figura 2.1, possui três colunas: nome_agência, número_conta e saldo. Seguindo a terminologia do modelo relacional, o autor trata os nomes dessas colunas como atributos. Para cada atributo há um conjunto de valores permitidos, chamados *domínio* do atributo em questão. Para o atributo nome_agência, por exemplo, o domínio é o conjunto de todos os nomes de agências. Supondo que D^1 denote este conjunto, D^2 denota o conjunto de todos os números de contas e D^3, o conjunto de todos os saldos. Qualquer linha de *conta* consiste necessariamente de uma 3-tupla (v_1, v_2, v_3) em que v_1é o nome da agência (isto é v_1, está no domínio D^1), v_2é um número de conta (Isto é, v_2 está no domínio D^2) e v_3 é um saldo (isto é, v_3 está no domínio D^3). Em geral a conta é um subconjunto de D^1 x D^2 x D^3.

nome_agência	número_conta	saldo
Lins	A-101	500
São Paulo	A-215	700
Marília	A-102	400
Santos	A-305	350

Figura 2.1 – Relação *conta*.

De acordo com Korth (1999), exige-se que, para todas as relações *r*, os domínios de todos os atributos de *r* sejam **atômicos**. Um domínio é atômico se elementos desse domínio são considerados unidades indivisíveis. Por exemplo, o conjunto dos inteiros é um domínio atômico, mas o conjunto de todos os conjuntos inteiros não é um domínio atômico.

Para Chays et al (2000), **relações** apresentam freqüentemente, a idéia de tabelas nas quais cada linha representa um dado sobre uma particular entidade e cada coluna representa um particular aspecto destes dados. Desta forma, para representar a *tupla* em uma tabela, pode-se utilizar a chave primária ou o conjunto de chaves primárias.

Encontra-se em Korth (1999, p.63) deixa clara a distinção entre **esquema de banco de dados**, ou seu esquema lógico, e uma **instância no banco de dados**, que é uma "foto" dos dados no banco de dados em determinado momento. O conceito corresponde em linguagem de programação seria a noção de variável. Um conceito de um **esquema de relação** corresponde, em linguagem de programação, à noção de definição de tipos. (KORTH, 1999) adota o uso de letras minúsculas para nomes de relações e nomes iniciados com uma letra maiúscula para esquemas de relações. Seguindo esta notação segue como exemplo o nome Esquema_conta para denotar o esquema de relação para a relação *conta*:

Esquema_conta (nome_agência, número_agência, saldo)

Denota-se o fato de *conta* ser uma relação em Esquema_conta por:

Conta(Esquema_conta)

Em geral, um esquema de relação compreende uma lista de atributos e seus domínios correspondentes, segundo (KORTH, 1999).

Para Korth (1999), o conceito de **instância de relação** corresponde, em linguagem de programação, ao valor de uma variável. O valor de uma dada variável pode mudar ao longo do tempo; de modo similar, o conteúdo de uma instância de relação pode mudar com o

tempo, quando esta relação é atualizada. Entretando, usa-se freqüentemente "*relação*", quando na realidade refere-se à "*instância de relação*".

Uma instância de relação em um dado tempo representa o estado atual da relação e reflete somente as *tupla*s válidas que representa um estado particular do mundo real. Em geral, como o estado do mundo real se altera, a relação também é transformada para outro estado. Entretanto, o esquema de relação é relativamente estático e não se altera, exceto esporadicamente (ELMASRI e NAVATHE, 2005, p.92).

Para qualificar um SGBD relacional genuíno, o sistema deve possuir pelo menos as seguintes propriedades (ELMASRI e NAVATHE, 2005):

a) Armazenar os dados com relação de tal maneira que cada coluna seja identificada independentemente de seus nomes sem que a ordenação das linhas seja importante;

b) As operações disponíveis para o usuário e as operações usadas internamente pelo sistema devem ser operações relacionais verdadeiras (disponíveis para gerar novas relações a partir de relações existentes);

c) Suportar pelo menos uma variante de operação de junção (*JOIN*).

Outro item seria a *restrição de integridade* que restringe os possíveis valores do estado do banco de dados, bem como reflete exatamente o mundo real da entidade que está sendo modelada, onde será detalhado a seguir.

2.1.3. Regras de Integridade

Em Korth (1999), temos que as regras de integridade fornecem a garantia de que mudanças feitas no banco de dados por usuários aurtorizados não resultem em perda da consistência dos dados. Assim, as regras de integridade protegem o banco de dados de danos acidentais. Essas regras possuem a seguinte forma conforme Korth (1999):

- **Declaração de chaves**: a determinação de certos atributos, como chave candidata para um dado conjunto de entidades. O conjunto de inserções e atualizações válidas é restrito àquelas que não criem duas entidades com o mesmo valor de chave candidata.
- **Forma de um relacionamento**: muitos para muitos, um para muitos, um para um. O conjunto de relacionamento um para um ou um para muitos restringe o conjunto de relacionamentos válidos entre os diversos conjuntos de entidades.

Segundo Korth (1999) os tipos de restrições são descritas da seguinte maneira:

a) **Restrições de Domínio**: são as mais elementares formas de restrições de integridade. Elas são facilmente verificadas pelo sistema sempre que um novo item de dado é incorporado ao banco de dados. É possível que diversos atributos tenham um mesmo domínio. Por exemplo, os atributos *nome_cliente* e *nome_empregado* podem ter o mesmo domínio (o conjunto de todos os nomes de pessoas). Entretanto, os domínios de *saldo* e *nome_agência* certamente serão distintos. A cláusula ***check*** da SQL-92 permite modos poderosos de restrições de domínios que a maioria dos sistemas de tipos das linguagem de programação não permite. Especificamente, a cláusula ***check*** permite ao projeto do esquema determinar um predicado que deva ser satisfeito por qualquer valor designado a uma variável cujo tipo seja o domínio. Por exemplo uma cláusula ***check*** pode garantir que o domínio relativo ao turno de trabalho de um operário contenha somente valores maiores que um dado valor (turno mínimo). A cláusula check pode também ser usada para restringir valores nulos em um domínio (*not null*).

b) **Integridade Referencial**: freqüentemente, deseja-se garantir que um valor que aparece em uma relação para um dado conjunto de atributos também apareça para um certo conjunto de atributos de outra relação. Essa condição é chamada *integridade referencial*. Na integridade referencial em SQL é possível definir chaves primárias

(*primary key*), e chaves estrangeiras (*foreign key*) como parte do comando *create table* da SQL: a cláusula ***primary key*** do comando *create table* inclui a lista de atributos que constituem a chave primária. A cláusula ***unique*** do comando *create table* inclui a lista dos atributos que constituem uma chave candidata. A cláusula ***foreign key*** do comando *create table* inclui tanto a relação dos atributos que constituem a chave estrangeira quanto o nome da relação à qual a chave estrangeira faz referência. Quando uma regra de integridade referencial é violada, o procedimento normal é rejeitar a ação que ocasionou esta violação. Entretanto, a cláusula relativa à foreign key em SQL-92 pode especificar que, se uma remoção ou atualização na relação que ela faz referência violar uma regra de integridade, então, em vez de rejeitar a ação, executam-se passos para modificação da tupla na relação que contem a referência, de modo a garantir a regra de integridade.

Date (2000), divide as restrições de integridade em quatro categorias:

a) **Restrição de Tipo**: especifica os valores válidos para um determinado tipo (ou domínio), e é verificada durante invocações do **seletor** correspondente.
b) **Restrição de Atributo**: especifica os valores válidos para um determinado atributo, e nunca deve ser vioada.
c) **Restrição de Variável de Relação**: especifica os valores válidos para uma determinada variável de relação, e é verificada quando essa variável de relação é atualizada.
d) **Restrição de Banco de Dados**: especifica os valores válidos para um determinado banco de dados, e é verificada no instante do *COMMIT*.

Segue as características das restrições de integridade semântica**.**

2.1.4. Restrições de Integridade Semântica

> ***Restrições de Integridade Semântica***: são restrições gerais sobre os valores do estado do banco de dados, expressos em algumas restrições de linguagens específicas. Por exemplo, estes devem expressar regras de negócio da organização de onde os dados estão sendo modelados, tais como um requerimento onde o salário do chefe de departamento deve ser maior do que qualquer outro membro do departamento. (ELSMASRI e NAVATHEE, 2005)

A integridade semântica é a garantia de que o estado dos dados do banco de dados está sempre coerente com a realidade para a qual o mesmo foi projetado e criado (ELMASRI e NAVATE, 2005). A inexistência ou a falta de gerenciamento de integridade semântica pode causar, por exemplo, violação de domínio, ausência de valor em atributos e relacionamentos incorretos.

As subseções anteriores apresentaram algumas características relevantes para o enfoque deste trabalho as quais são abordadas na etapa experimental do teste estrutural de ABDR. Segue agora, o funcionamento da SQL (*Structured Query Language*).

2.1.5. SQL (*Structured Query Language*)

Esta subseção apresenta as principais características da Linguagem SQL aplicada na Linguagem Java a qual foi utilizada como uma aplicação para estudo de casos deste trabalho.

Atualmente, a SQL (*Structured Query Language*) é uma linguagem padrão utilizada para definir e manipular dados em um banco de dados, sendo utilizada em grande parte dos sistemas de banco de dados comerciais.

Em Korth (1999), p.104, no histórico da linguagem da SQL, é que a versão original foi desenvolvida pela IBM e essa linguagem, originalmente foi chamada de Sequel no início dos anos 70. A Sequel foi evoluindo e seu nome foi mudado para SQL (*Structured Query Language* – Linguagem de Consulta Estruturada).

A linguagem SQL possue diversas partes, conforme Korth (1999), dentre elas tem-se:

a) **Linguagem de definição de dados** (*Data-definition Language* - DDL). A SQL DDL proporciona comandos para a definição de esquemas de relações, criação de índice e modificação nos esquemas de relações.

b) **Linguagem interativa de manipulação de dados** (*Data-manipulation Language* – DML). A SQL DML abrange uma linguagem de consulta baseada tanto na álgebra relacional quanto no cálculo relacional de tuplas. Engloba também comandos para inserção, exclusão e modificação de tuplas no banco de dados.

c) **Incorporação DML** (Embedded DML). A forma de comandos SQL incorporados foi projetada para aplicação em linguagens de programação de uso geral, como PL/I, Java, C.

d) **Definição de visões.** A SQL DDL possui comandos para definição de visões.

e) **Autorização.** A SQL DDL engloba comandos para especificação de direitos de acesso a relações e visões.

f) **Integridade.** A SQL DDL possui comandos para especificação de regras de integridade que os dados que serão armazenados no banco de dados devem satisfazer. A atualizações que violarem as regras de integridade serão desprezadas.

g) **Controle de transações.** A SQL inclui comandos para especificação de iniciação e finalização de transações. Algumas implementações também permitem esplicitar bloqueios de dados para controle de concorrência.

A **Linguagem de controle de dados** (DCL) controla os aspectos de licenças de usuários para controlar quem tem acesso para ver ou manipular dados dentro do banco de dados.

Para o experimento executado neste trabalho, apenas os comandos **DML** é que serão utilizados para o enfoque de teste, sendo estes os comandos responsáveis pela definição e uso das tabelas de banco de dados de uma aplicação, como ***INSERT***, ***UPDATE***, ***DELETE*** e ***SELECT***.

Alguns exemplos de comandos **DDL** utilizados são (ORACLE, 1999):

- *ALTER*, *CREATE*, *DROP* e *RENAME*, que definem dados das relações;
- *CONNECT*, *GRANT*, *LOCK* e *REVOKE*, que controlam o acesso aos dados dos esquemas da base de dados.

Alguns exemplos de comandos **DCL** apresentados em Date (2000):

- *GRANT* – autoriza ao usuário executar ou setar operações
- *REVOKE* – remove ou restringe a capacidade de um usuário de executar operações

Alguns exemplos de comandos DML utilizados são (ORACLE, 1999):

- *DELETE*, *INSERT* e *UPDATE*, usados para manipular os dados das relações;
- *CLOSE*, *FETCH*, *OPEN* e *SELECT*, usados para recuperar os dados das relações;
- *COMMIT*, *ROLLBACK*, *SAVEPOINT* e *SET TRANSACTION*, usados para processar as transações das relações da base de dados;
- *DESCRIBE*, *EXECUTE* e *PREPARE*, usados para aplicação com SQL dinâmico, permitindo criar consultas em tempo de execução.

A SQL também inclui mecanismos para especificar e obrigar restrições de segurança, para obrigar restrições de integridade e embutir declarações dentro de outras linguagens de programação de alto nível. SQL permite ao usuário expressar operações para consulta e modificar o banco de dados em uma maneira de alto nível, expressando o que deve ser feito, em lugar de como deve ser feito. Desse modo, SQL é largamente usada em sistemas de banco de dados comercial.

2.2. Teste de Aplicação de Banco de Dados Relacional

Esta seção apresenta a composição das etapas de teste estrutural de aplicação de banco de dados relacional. Inicialmente são abordados o teste de unidade e o teste de integração.

- Teste de Unidade

Objetiva-se no **Teste de Unidade** a execução e exame da menor parte do software, que pode ser uma função (de uma linguagem de programação procedimental). Os resultados baseados na especificação são comparados com os resultados obtidos durante a execução de um caso de teste para saber se algum defeito foi revelado. Isto é feito para cada unidade testada isoladamente.

Para executar cada unidade isoladamente, pode ser necessária a construção de *drivers* e *stubs* de teste. O *driver* recebe os dados de teste, passa-os como parâmetros para a unidade de programa que está sendo testada e mostra os resultados produzidos para que o testador os avalie. O *stub* serve para substituir as partes subordinadas às unidades em teste. Esses *stubs* são geralmente unidades que simulam o comportamento de unidades reais de programas através de um mínimo de computação ou manipulação de dados (VILELA, 1998). Na Figura 2.2 é mostrada uma unidade em teste com seus respectivos *stubs* e *driver*.

A criação de *stubs* e *drivers* muitas vezes é uma tarefa demorada, mas que necessita de atenção, pois é de grande importância para o teste de unidade.

No **Teste de Unidade** o programa pode ser avaliado segundo a visão funcional do *software* ou segundo a visão da estrutura de implementação (visão estrutural) (SPOTO, 2000).

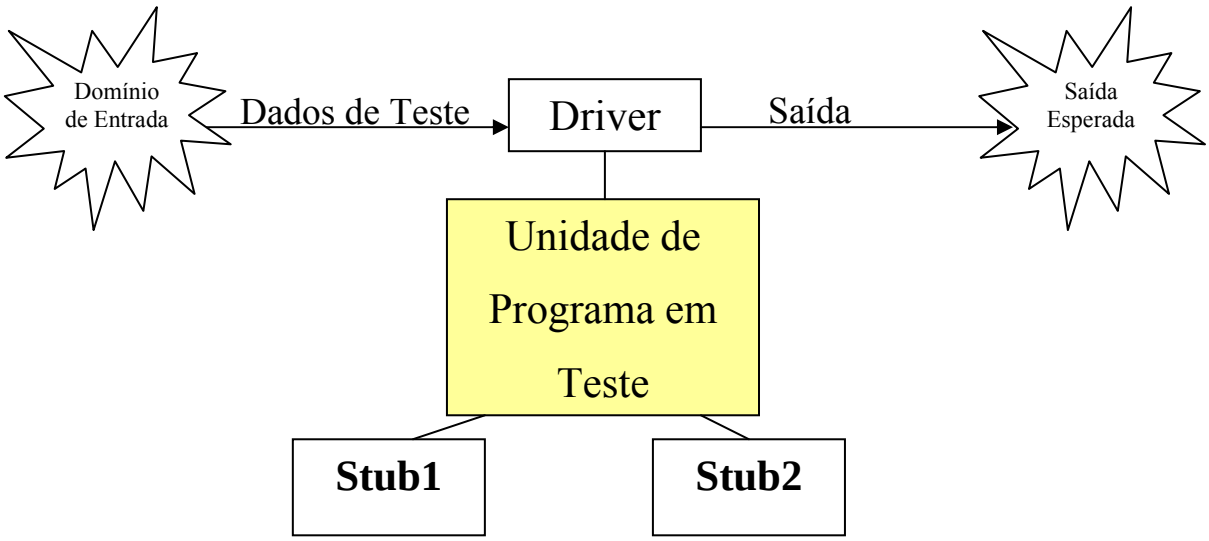


Figura 2.2: Ambiente para o teste de unidade

As técnicas de teste funcional, estrutural e baseada em erros proporcionam diferentes tipos de descobertas de defeitos, sendo assim, elas são complementares. Para um teste com maior abrangência recomenda-se utilizar as técnicas para que o teste seja o mais completo possível, tornando o software em teste mais confiável após estas técnicas terem sido exercitadas. O teste de unidade vem sendo historicamente aplicado, principalmente à técnica de teste estrutural, segundo Spoto (2000). O teste de unidade inicia-se na etapa de implementação, podendo se estender até a etapa do teste de integração, que se inicia na implantação do sistema. Após o teste de integração são realizados o teste de sistema e o teste de aceitação do usuário (PRESMAN, 1995).

- Teste de Integração

O **Teste de Integração** é geralmente aplicado após todas as unidades que compõem um programa terem sido testadas isoladamente, ou seja, após ter sido executado o teste de unidade. O teste de integração pode ser visto como uma técnica sistemática para a construção da estrutura do software procurando revelar defeitos associados à interação entre as unidades

de programa ou entre os módulos dos programas. O teste de integração cuida das questões associadas aos duplos problemas da verificação e construção de programas (PRESSMAN, 1995). Visa também a identificar defeitos na interação entre módulos e constitui uma técnica para integrá-los (VILELA, 1998).

Algumas características do teste de unidade podem ser aproveitadas e adaptadas para o teste de integração.

Pressman (1995) recomenda a utilização de uma abordagem incremental para o teste de integração, pois, o programa é construído e testado em pequenos segmentos, nos quais defeitos são detectados e corrigidos mais facilmente, além das interfaces serem exercitadas de maneira mais completa. Já na maneira não incremental, em uma abordagem conhecida como "integração *big-bang*" todas as unidades são colocadas para interagir juntas, de uma só vez, e o programa é testado como um todo, onde geralmente, o resultado é uma situação caótica, dificultando a correção, devido à dificuldade de identificar a origem do problema. No modo incremental, atualmente, existem 3 tipos de integração: *top-down*, *botton up* e sanduíche (PRESMAN, 1995).

Segundo Spoto (2005), a dependência de dados é um aspecto fundamental associado à interação entre Unidades de Programas e Módulos da Aplicação. As dependências de dados existentes entre UPs de programas convencionais são ocasionadas pelas variáveis globais ou por variáveis passadas por parâmetros por comandos de chamadas. ABDRs possuem dependências de dados baseadas nos comandos de chamada, como acontece nos programas convencionais e dependências de dados baseadas nas tabelas da base de dados. As dependências de dados entre UPs e Módulos de Aplicação de uma ABDR dão ensejo a novas abordagens de integração (SPOTO, 2005).

Em muitos casos os programas são testados isoladamente à medida que os módulos vão sendo concluídos, com a finalidade de confirmar se o módulo foi codificado

corretamente. Após a conclusão da integração entre os módulos, inicia-se o teste entre os grupos de programas, já integrados, até concluir todo o "teste de sistemas". No teste de sistema, a integração é realizada para integrar as interfaces e assegurar que os módulos estejam se comunicando da maneira esperada. Em seguida, o software é explorado como forma de detectar suas limitações e medir suas potencialidades. Em um terceiro nível, sistemas completos são, por fim, submetidos a um "teste de aceitação" para verificar a possibilidade de implantação e uso, geralmente feita pelo cliente ou usuário final.

2.2.1. Teste em Banco de Dados – definições e terminologia

Segue alguns conceitos de Aranha et. al. (2000):

a) **Caso de teste** consiste na associação de uma “query” com o resultado esperado. Por exemplo, ao se atualizar um determinado atributo de uma relação espera-se que algumas tuplas sejam afetadas na relação; o resultado esperado é a especificação das tuplas afetadas e os respectivos valores.

b) O **teste de relação** é o exercício de uma unidade da base de dados – uma relação – para detecção de defeitos na definição das estruturas dos atributos e de suas restrições. O **teste de relacionamento** baseia-se no projeto da base de dados e tem como objetivo revelar problemas nos relacionamentos entre as relações, através do exercício de chaves.

c) **Critérios de teste** estabelecem requisitos a serem satisfeitos pela execução do teste e podem auxiliar na seleção de dados de testes e na avaliação da qualidade de teste.

d) **Satisfazer um critério** significa atender aos requisitos de teste estabelecidos pelo critério.

e) **Elementos requeridos de um critério**, são os elementos da base de dados que devem ser exercitados para que se considere satisfeito.

f) **Cobertura de um critério** é uma medida do porcentual dos elementos requeridos que foram efetivamente exercitados em relação ao número total de elementos requeridos pelo critério. É uma medida de avaliação da qualidade de um teste segundo um critério.

Deve-se atentar para o estado do banco de dados antes e depois da execução de um caso de teste (CHAYS et al., 2000). Isso se faz necessário, pois uma falha causada por um caso de teste anterior, pode, por exemplo, modificar o estado da relação de forma a tornar o banco de dados inconsistente, influenciando o resultado do caso de teste atual. As alterações que ocorrem no banco de dados, a qualidade da informação armazenada e sua confiabilidade são itens imprescindíveis para análise do banco de dados.

Segundo Batista (2003), para colocar o banco de dados no estado desejado, faz-se necessário incluir, excluir ou alterar dados do banco, mantendo somente dados válidos de acordo com os domínios, com as relações e com as restrições.

As principais técnicas de testes existentes, atualmente, são:

- ***Técnica de Teste Funcional (caixa preta)***: os requisitos de testes são obtidos a partir da especificação de programas. Tem esse nome pelo fato de tratar o software como uma caixa, na qual o conteúdo é desconhecido e só é possível visualizar o lado externo. O comportamento somente pode ser determinado estudando-se suas entradas e saídas relacionadas. O testador está preocupado somente com a funcionalidade, e não com a implementação do *software* (SOMMERVILLE, 2003);

- ***Técnica de Teste Baseado em Defeitos***: esta técnica de teste utiliza informações sobre os defeitos típicos identificados durante o processo de desenvolvimento de *software* e sobre os tipos específicos de defeitos que se desejam revelar (DEMILLO, 1978). Há dois critérios típicos baseados em defeitos: Semeadura de Erros (*Error Seeding*) e Análise de Mutantes (*Mutation Analysis*);

- ***Técnica de Teste Estrutural***: será descrita com mais detalhes a seguir, tendo em vista que esta técnica é o enfoque principal deste trabalho.

2.2.2. Técnica de Teste Estrutural

Na ***Técnica de Teste Estrutural***, também conhecida como caixa branca, o teste é baseado em programas, pois, leva em consideração, os aspectos de implementação do software na determinação dos requisitos de teste. Geralmente, a maioria dos critérios desta técnica se apóia em uma representação de programa, conhecida como grafo de fluxo de controle ou grafo de programa, que é uma notação para representar o fluxo de controle lógico de uma unidade de programa e consiste basicamente de um grafo direcionado, estabelecendo uma correspondência entre os nós e blocos indicando possíveis fluxos de controle entre blocos através dos arcos. A partir do grafo de programa podem ser escolhidos os elementos que devem ser exercitados, caracterizando assim o teste estrutural.

2.2.2.1. Teste estrutural Baseado no Fluxo de Controle

A análise de fluxo de controle visa a exercitar os comandos que abrangem tanto a parte computacional de um programa como a parte condicional do programa. Para isso os critérios baseados em fluxo de controle foram os primeiros critérios na literatura a abranger o teste estrutural. A seguir são apresentados os principais critérios utilizados para a análise de fluxo de controle a partir de um grafo de programa em teste:

- Critério ***todos os nós*** - todos os comandos que fazem parte de cada nó do grafo de controle da unidade em teste devem ser executados pelo menos uma vez;
- Critério ***todos os arcos*** - todos os comandos de transferência condicional devem ter suas condições executadas pelo menos uma vez;
- Critério ***todos os caminhos*** - todos os comandos que fazem parte de cada caminho do grafo de controle devem ser executados pelo menos uma vez.

2.2.2.2. Teste Estrutural Baseado em Fluxo de Dados

Na década de 1970 surgiram os **critérios baseados em análise de fluxo de dados (**HERMAN, 1976) em que a característica comum é requerer que sejam testadas as interações que envolvam definições de variáveis e subseqüentes referências a essas definições, ou seja, o uso destas variáveis. Sendo assim, consiste em enfocar a atribuição de valores às variáveis e o uso posterior destes valores, estabelecendo que a ocorrência de uma variável pode ser de dois tipos: definição e uso. O teste baseado unicamente no fluxo de controle não é eficaz em revelar a presença de erros simples, portanto, foi um incentivo para a introdução dos critérios baseados em fluxo de dados, onde visa a fornecer uma hierarquia de critérios entre os critérios *todos os arcos* e *todos os caminhos*, proporcionando um teste mais rigoroso. Neste sentido, destacam-se os critérios de fluxo de dados introduzidos por Rapps e Weyuker, nos anos de 1980, como: *todas as definições*, *todos os usos* e *todos os du-caminhos*.

De acordo com Huang (1975 apud SPOTO, 2000), no teste estrutural, um programa *P* pode ser decomposto em um conjunto de blocos distintos de comandos. Um bloco de comandos é a seqüência de um ou mais comandos com a propriedade que, sempre que o primeiro comando do bloco é executado, todos os demais comandos do bloco também são executados e não existem desvios para o meio do bloco.

Um programa *P* é representado por um grafo de fluxo de controle (ou grafo de programa) $G= (N,A, e, s)$ onde *N* é o conjunto de nós, *A* é o conjunto de arcos (ou arestas), *e* é o único nó de entrada e *s* é o único nó de saída. Todo grafo de programa é um grafo dirigido conexo. Um nó $n \in N$ representa uma instrução simples (comando) ou uma seqüência de instruções executadas como um bloco de comandos. Cada arco $a \in A$ é um par ordenado (*n, m*) de nós *N* que representa uma possível transferência de controle do nó *n* para o nó *m* (HERMAN, 1976).

Um **grafo de fluxo de controle (CFG)** de um programa é um grafo direcionado que representa o controle de estrutura de um programa (OFFUT e PAN, 1991).

Um *caminho* do programa é representado por uma seqüência finita de nós (n_1, n_2, ..., n_k), $k >= 2$, tal que, para todo nó n_i, $1 <= i <= k-1$, existe um arco (n_i, n_{i+1}) para $i = 1, 2, ..., k-1$. Um *caminho completo* é um caminho cujo nó inicial é o nó de entrada e cujo nó final é o nó de saída. Um *caminho simples* é um caminho cujos nós, exceto possivelmente o primeiro e o último, são distintos. Um caminho é *livre* de laço se todos os nós pertencentes a ele são distintos (RAPPS e WEYUKER, 1985).

Considerando Offut e Pan (1991), se os dados de entrada satisfazem a condição de caminho existente, o controle de caminho é também uma **execução de caminho** e pode ser usado para testar o programa. Se a condição de caminho não pode ser satisfeita, o controle de caminho é dito **infactível (não executável)**.

A análise estática do programa fornece informações sobre as ações executadas a respeito das variáveis do programa e efeitos de tais ações nos vários pontos do programa Hectch (1977 apud SPOTO, 2000). Ainda Spoto (2000) considera, em geral, que uma variável possa sofrer as seguintes ações no programa: (*d*) definição, (*i*) indefinição; ou (u) uso. Considerando os critérios de teste estrutural aplicado na unidade de um programa, definidos por Rapps e Weyuker (1985), bem como Maldonado (1991) uma ocorrência de variável é uma *definição* se ela está: i) ao lado esquerdo de um comando de atribuição; ii) em um comando de entrada ou iii) em chamadas de procedimentos como parâmetro de saída (MALDONADO, 1991). A ocorrência de uma variável como *uso* se dá quando a referência a esta variável, em um comando executável, não a estiver definindo, ou seja, em uma *recuperação* de um valor em uma posição de memória associada a esta variável.

Rapps e Weyuker (1985) definem dois tipos de usos de uma variável: o primeiro *c-uso* (uso computacional), que afeta diretamente uma *computação* que está sendo realizada ou

permite observar o valor de uma variável que tenha sido definida anteriormente (nesses casos o *uso* está associado a um nó do grafo do programa). O outro tipo de uso é o *p-uso* (uso predicativo), que afeta diretamente o *fluxo de controle* do programa (este *uso* está associado a um arco do grafo). Uma variável está *indefinida* quando, ou não se tem acesso ao seu valor, ou sua localização deixa de estar definida na memória.

A classe de critérios baseados em fluxo de dados utiliza informações do fluxo de dados do programa para derivar os requisitos de teste. Esses critérios requerem que sejam testadas as interações que envolvem definições de variáveis e referências a essas definições (RAPPS e WEYUKER, 1985).

Para derivar os requisitos de teste requeridos por esses critérios é necessário adicionar ao grafo de programa informações sobre o fluxo de dados, caracterizando o **Grafo Def-Uso** (*Def-Use Graph*) definido por Rapps e Weyuker (1985). Neste grafo são exploradas as associações entre a definição e o uso das variáveis determinando os caminhos a serem exercitados.

A partir dos critérios de Rapps e Weyuker, no início dos anos de 1990, (MALDONADO, 1991) introduziu-se a família de critérios Potenciais-Usos e a correspondente família de critérios Executáveis, obtida pela eliminação dos caminhos e associações não executáveis. Esses se baseiam nas associações entre uma definição de uma variável e seus possíveis subseqüentes usos para a derivação de casos de teste.

As variações da família do critério de *Potenciais-Usos* são: *todos os potenciais-usos, todos os potenciais-usos/du* e *todos os potenciais-du-caminho*.

Estes critérios requerem associações que implicam no exercício de caminhos entre a definição de uma variável e um possível (potencial) uso desta variável (CHAIN, 1991).

A família de critérios Potenciais Usos tem como característica básica o fato de requerer associações independentemente da ocorrência explícita de uma referência (uso) a uma determinada definição.

Considera Vincenzi (1998), que de modo semelhante aos demais critérios de fluxo de dados, os *potenciais-usos* podem utilizar o Grafo-Def-Uso como base para o estabelecimento dos requisitos de teste. Na verdade, basta estender o grafo de programa para que cada nó do grafo passe a conter informações a respeito das definições que ocorrem em cada nó (MALDONADO, 1991).

Com o conceito de Potencial Uso, busca-se explorar todos os possíveis efeitos a partir de uma mudança de estado do programa em teste. Tanto os critérios Potenciais-uso (MALDONADO, 1991) como os definidos por Rapps e Weyuker (1985) podem ser usados para a etapa de teste de unidade para avaliar as variáveis de programa de uma ABDR.

A seguir serão introduzidos os conceitos de teste estrutural de ABDR definidos por Spoto (2000), que são os principais enfoques deste trabalho.

3. TESTE ESTRUTURAL DE APLICAÇÃO DE BANCO DE DADOS RELACIONAL

Segundo Spoto (2000), uma **Aplicação de Banco de Dados Relacional** é um conjunto de Módulos de Programas: ABDR = $\{Mod_1, Mod_2,..., Mod_m\}$, $m \geq 1$, onde cada Módulo de Programa Mod_i pode ser escrito em linguagem *C*, *Pascal*, *Fortran*, *Ada*, *PL/I* ou outras linguagens, dependendo do Sistema Gerenciador de Banco de Dados (SGBD) que hospeda a linguagem de manipulação da base de dados (DML). Spoto (2000) define que cada Módulo de Programa é composto por vários procedimentos que denominamos de Unidades de Programas: Mod = $\{UP_1, .., UP_n\}$, para $n \geq 1$. Esses programas interagem com as tabelas (t_1, t_2,..) da Base de Dados que compõem a Aplicação. Para satisfazer todas as etapas de teste de uma ABDR, foram criados dois tipos de Fluxo de Dados, baseados em Harrold e Rothermel (HARROLD, 1994 apud SPOTO, 2000) "fluxo de dados intra-modular" – fluxo de dados dentro de um programa; e "fluxo de dados inter-modular" – fluxo de dados entre programas distintos da ABDR.

As Aplicações de Banco de Dados Relacional (ABDR) são formadas por programas de linguagens convencionais como *C*, *Pascal*, *Java*, *Cobol*, *ADA*, *PL/I* e outras mais, que permitem o uso de comandos SQL embutidos em seu código (SPOTO,2000).

Em Spoto (2000) a aplicação de Banco de Dados é composta por um ou vários programas (descritos em linguagens de programação procedimental) que suportam comandos da linguagem SQL. Para isso, o SGBD deve possuir um pré-compilador que processa o programa fonte gerando um novo programa modificado que será, então, submetido à compilação e geração do programa executável. As linguagens de programação mais comuns como *C*, *Pascal*, *Java*, *Fortran*, *Cobol*, e *PL/I* são aceitas pela maioria dos SGBDs relacionais.

O termo linguagem hospedeira, segundo Spoto (2000), é usado para representar a linguagem existente nos SGBDs que possibilita o uso de comandos de SQL; programas hospedeiros são os programas da aplicação que hospedam a linguagem SQL.

Manilla (1989) descreve uma técnica para a abordagem de teste estrutural aplicado a sistemas de Banco de Dados Relacional, sendo que esta técnica gera uma base de dados mínima a partir de uma base mais completa que é, segundo o autor, suficiente para ser usada como uma massa de teste para exercitar as dependências entre as possíveis classes de consultas (select, project, join) de um programa em teste. Muito importante para a geração dos casos de testes em ambiente de Banco de Dados.

Em Aranha et al. (2000), é apresentada uma ferramenta (RDBTool) que fornece recursos para análise estática do esquema da base de dados, validação estática e dinâmica dos dados da base, análise de cobertura para cada critério utilizado e um guia para auxiliar o testador em suas atividades. A idéia é executar operações especificadas do esquema, segundo os critérios de teste, visando à ocorrência de falhas (que indicam defeitos no esquema da base de dados). Apresenta ainda técnicas para testar os esquemas das bases de dados que exercita os atributos e as restrições de integridade da base de dados através de *queries*. São definidos os critérios *todos os tipos de definição*, *todas associações de tipos de definição* e o *critério pelo menos um tipo de definição*.

Batista (2003) propõe um sistema, denominado DBValTool(*DataBase Testing and Validation Tool)*, que tem a função de testar uma base de dados, visando detectar possíveis erros na criação dos esquemas relacionais, antes de passar para etapas seguintes do processo de desenvolvimento.

Conforme Chays et al (2000) uma atenção relativamente pequena que tem sido dada para o desenvolvimento de técnicas sistemáticas para assegurar a exatidão de programas de aplicação de banco de dados. Dados os papéis críticos destes sistemas em uma sociedade

moderna, há claramente uma necessidade para novas abordagens em avaliar a qualidade dos programas de aplicação de banco de dados. Em Chays et al (2000) apresentada uma abordagem baseada na indicação do testador de dados típicos para cada atributo na base de dados e em Chays et al (2003) é apresentada uma ferramenta que automatiza esta abordagem, auxiliando a aplicação de casos de teste e a checagem se as saídas e o novo estado da base de dados estão consistentes com o comportamento esperado.

São de extrema importância técnicas específicas para testar as bases de dados, principalmente antes de serem instanciadas, pois, os bancos de dados desempenham um papel importante nas organizações, armazenando, disponibilizando e protegendo uma numerosa quantidade de dados, além de garantir a consistência e integridade destes.

Há características similares entre teste de programas convencionais e teste estrutural de programas de ABDR (Aplicações de Banco de Dados Relacional), adaptando nesta, comandos SQL, variáveis de tabela e variáveis *host* (quando estas diferem das variáveis de programa).

Utilizando a abordagem descrita em Chays et. al. (2000), onde uma aplicação de banco de dados, como qualquer outro programa, pode ser vista como uma computação, função de um espaço de entrada *I* para um espaço de saída *O*. Esta especificação pode ser expressa como uma função (ou, mais geral, uma relação) de *I* para *O*. Assim pode-se testar uma aplicação de banco de dados pela seleção de valores de *I*, executando a aplicação sobre eles, e verificando se o resultado de *O* está de acordo com a especificação. Em Chays et al. (2003), os autores apresentaram uma ferramenta para auxiliar a aplicação de casos de teste e a checagem se as saídas e o novo estado da base de dados estão consistentes com o comportamento esperado.

Spoto (2000) apresenta uma abordagem mais completa de teste estrutural de banco de dados relacional e propõe técnicas apropriadas para o teste de Aplicação de Banco de

Dados Relacional (ABDR) para variáveis tabela presentes nestes programas, que utilizam a Linguagem SQL. Esta abordagem propõe para o teste estrutural de programas ABDR, com dois tipos de Fluxo de Dados: intra-modular (aplica-se ao teste de unidade e ao teste de integração das unidades de um programa) e inter-modular (aplica-se à integração dos programas quem compõem uma aplicação). Tendo em vista que a ABDR está implementada em Linguagem Java, será abordado o primeiro fluxo de dados como intra-classe e o segundo fluxo de dados como inter-classe; as Unidades de Programas (UPs), correspondem aos Métodos e os Módulos de Programa correspondem às Classes.

As variáveis tabelas possuem um enfoque mais amplo por serem variáveis persistentes cuja definição só é concretizada quando for validada a transação dos dados (utilizada pelo comando *COMMIT*) (SPOTO et al, 2005).

Devido à existência dos comandos de SQL em programas de ABDR, a definição de grafo de programa foi alterada para acomodar os comandos executáveis da SQL em nós especiais, um em cada nó. A representação gráfica de uma unidade de programa é a mesma adotada em Maldonado (1991), e estendida para ABDR com o uso de SQL. Na representação gráfica, os comandos da Linguagem Java e os comandos declarativos da SQL podem ser acomodados em blocos de comandos. Os comandos executáveis da Linguagem SQL (*INSERT, DELETE, UPDATE, SELECT, COMMIT, ROLLBACK*, entre outros) são acomodados isoladamente. Como notação gráfica adotaram-se nós circulares para representar os blocos de comandos e nós retangulares para representar os comandos executáveis da SQL, podendo ser visualizados no grafo da Figura 3.l, em que o método *inserirDados()* os nós retangulares 62 e 78 representam os comandos da SQL *insert* e *commit* respectivamente. As setas, denominadas de arcos, representam possíveis transferências de controle entre os nós (SPOTO, 2000).

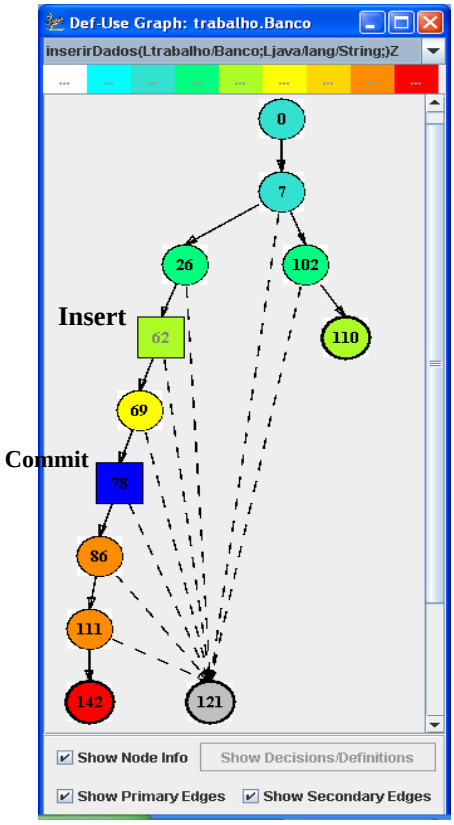


Figura 3.1 – Grafo do Método *inserirDados()*

Ainda em Spoto (2000), o grafo de fluxo de controle que representa uma UP de um programa da ABDR é denotado por G(UP) = (N^{BD}, E, n_{in}, n_{out}), $N^{BD} = N_h \cup N_S$, onde N_h é o conjunto de nós da linguagem hospedeira (*C*, *Pascal*, *Java* etc), representados no grafo por nós arredondados, e N_S é o conjunto de nós tal que cada nó corresponde a um comando executável da SQL, representados por nós retangulares. E é o conjunto de arcos, $E \subseteq N^{BD} \times N^{BD}$. Os nós $n_{in} \in N_h$ são os nós de entrada e $n_{out} \in N^{BD}$ são os nós de saída do grafo de programa. Em programas escritos em Java, o fluxo de controle para tratamento de erros, nos nós da SQL é ocasionado pelo comando ***catch* (*SQLException* <*variavel*>)**.

Um caminho é uma seqüência finita de nós *(n1, n2,..,nk), k ≥ 2,* tal que, para todo nó n_i,*1 ≤ i ≤ k-1* existe um arco $(n_i, n_i+1) \in E$ que vai de n_i para n_{i+1}. O grafo de programa estabelece uma correspondência entre os nós de N_h e N_S indicando os possíveis fluxos de

controle entre os nós através dos arcos (SPOTO et al, 2000). Apenas os comandos executáveis da **SQL** como: *a) INSERT, DELETE, UPDATE e SELECT* - de manipulação de dados; e *b) COMMIT* e *ROLLBACK* - de validação das tarefas; terão representação gráfica com nós retangulares (SPOTO, 1997). São estendidos para programas em Java a mesma notação de geração de nós retangulares para os comandos da SQL (mencionados neste parágrafo).

Como já comentado anteriormente, há características similares entre teste de programas convencionais e teste estrutural de programas de ABDR (Aplicações de Banco de Dados Relacional) com SQL, adaptando nesta, comandos SQL, *variáveis de tabela* e *variáveis host* (quando exigidas no programa), que não existem no teste de programas convencionais. Em Java a *variável host* será a mesma indicada como *variável de programa*.

No caso de programas em Java, a biblioteca padrão de persistência em banco de dados em Java é a **JDBC.** O sistema desenvolvido em Java abstrai o método através do qual é possível fazer uma conexão, pois as conexões são feitas através de uma ponte que implementa todas as funcionalidades que um banco de dados padrão deve nos fornecer. Por exemplo, toda conexão deve permitir executar código de atualização, pesquisa etc. Essa implementação precisa ser escolhida. Essa escolha não é feita programaticamente e sim basta usar uma ponte.

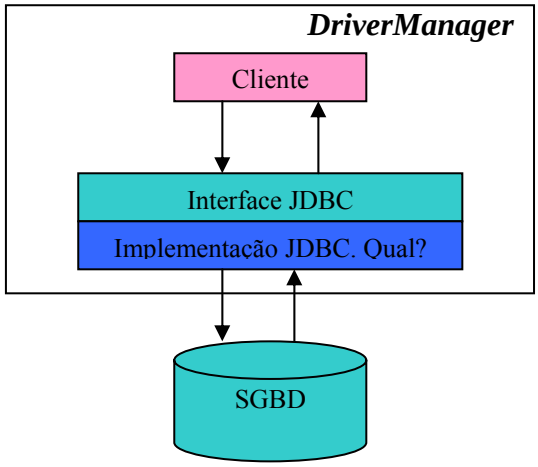


Figura 3.2 – Ponte de implementação entre o programa Java e SGBD

Veja na Figura 3.2 a ponte (implementação) entre o programa (cliente) e o banco de dados (SGBD). O serviço de encontrar uma ponte, ou seja, um *driver* certo é delegado a um controlador de *drivers*, que é o ***DriverManager*** () (HORTMANN e CORNELL, 2001, pp.180).

A análise de fluxo de dados baseia-se nos tipos de ocorrência das variáveis de um programa de aplicação. Em geral existem três tipos de variáveis:

a) ***Variáveis de programas***: variáveis definidas e usadas apenas na *linguagem.* No estudo de caso em questão, a Linguagem *Java*;

b) ***Variáveis host***: variáveis de ligação, são as variáveis que estabelecem os fluxos de dados entre a base de dados e o programa. Essas variáveis não serão necessárias para programas Java, pois, a própria variável de programa faz a ligação entre o programa e a SQL.

c) ***Variáveis tabela***: são definidas e usadas apenas nos comandos executáveis da SQL. Podem-se considerar tanto as *variáveis tabela persistentes* (gravadas em disco) como as *variáveis tabelas de visão* (residem na memória do computador durante as etapas de execução).

A *definição* de uma variável ocorre sempre que um valor é armazenado em uma posição de memória. Tais definições são tratadas apenas no momento da execução do programa que as gerou.

Seja $\boldsymbol{v} = \boldsymbol{P} \cup \boldsymbol{H}$ o conjunto de variáveis presentes em programas de ABDR com **SQL** embutida e seja **t** a variável de tabela (SPOTO, 2000).

As *variáveis tabela* são criadas pelo comando *CREATE TABLE* <*tabela*> e essa criação é realizada apenas uma vez durante o projeto de banco de dados. A partir de sua criação, o SGBD passa a controlar os acessos a tabelas da ABDR, permitindo que seus usuários possam modificá-las e / ou atualizá-las a partir dos programas de aplicação. Desse modo, adota-se a ocorrência de uma *variável tabela* ***t*** como sendo *definição* ou *uso* e

considera que toda a *variável tabela* referenciada por algum programa implica à ocorrência de uma definição anterior (até por um outro módulo de programa). Não há, portanto, nenhuma exigência sintática em declará-la antes de uma *definição,* ou antes, de um *uso*; para as demais variáveis pode ocorrer um erro de compilação ou uma "*anomalia*" (ORACLE, 2002; ELMASRI, 2005).

Segundo Spoto (2000), uma *variável tabela* é considerada definida quando o valor é armazenado em uma memória secundária modificando assim o estado da *variável tabela* de e_0 para e_1. Uma ocorrência de *variável tabela* em um programa é uma *definição* se ela estiver:

i) em uma cláusula INTO de um comando INSERT;

ii) em uma cláusula FROM do comando DELETE;

iii) do lado esquerdo da cláusula SET do comando UPDATE; e,

Esses comandos serão efetivados neste estudo de caso em Linguagem *Java* quando seu contexto estiver no comando *executeUpdate(query)*.

Apesar desses comandos da SQL caracterizarem a ocorrência de definição da *variável tabela*, as ocorrências acima *i*, *ii* e *iii* só serão efetivadas quando executadas junto com o comando *COMMIT*, quando o valor é armazenado em memória secundária. Para distinguí-las das demais definições (em memória interna), denomina-se de *definição persistente* de ***t*** sempre que ocorre uma alteração em seu estado físico (em memória secundária). Isto só é possível se houver um comando de manipulação da SQL seguido do comando *COMMIT*. Este trabalho focará mais as ocorrências de definição e uso persistente de uma *variável tabela*.

A ocorrência de uma variável é um **uso** quando a referência a essa variável não a estiver definindo. Uma ocorrência de uma *variável tabela* é um *uso* quando ela estiver:

i) na cláusula *FROM* dos comandos *SELECT* e *DELETE*;

ii) em uma cláusula INTO do comando INSERT;

iii) do lado esquerdo da cláusula *SET* do comando *UPDATE*

iv) na cláusula *WHERE* dos comandos executáveis *DELETE*, *UPDATE* e *INSERT*.

Portanto, a ocorrência das *variáveis tabela* precedidas com os comandos executáveis *INSERT*, *UPDATE* ou *DELETE* podem caracterizar tanto uma **definição** quanto um **uso** desta variável. O **uso** da *variável tabela* será considerado sempre no arco de saída do nó em que existir uma execução do comando SQL em que é efetuada a ocorrência do uso da *variável tabela*.

Um caminho $(i, n_1, n_2, .., n_k, j)$, $k \geq 0$, que contém uma *definição* da variável ***v*** no nó i e que não contenha nenhuma redefinição de ***v*** nos nós $n_1, n_2, ..., n_k$ é chamado de *caminho livre de definição c.r.a* ***v*** do nó i ao nó j e do nó i ao arco (n_k, j). Essa definição é válida para todos os nós N^{BD}, todos os arcos E e todas as variáveis de uma ABDR.

Dizemos que o caminho $(n_\lambda,..,n_i, n_1, n_2, .., n_k, n_j)$ é um caminho *livre de definição persistente* de $< n_\lambda, n_i>$ até o nó n_k, se não existe outro par de nós $<n_q, n_m>$ onde ocorre uma *redefinição persistente* de ***t*** em $<n_q, n_m>$ do nó n_i até o nó n_k onde o par $< n_\lambda, n_i>$ antecede o par $<n_q, n_m>$. Lembrando que na ausência de um comando *COMMIT*, após um comando executável da SQL, adota-se que o *COMMIT* está no próximo nó do grafo (sem ser a saída de exceção), podendo ser um *AUTOCOMMIT* ativo no programa em Linguagem Java.

De acordo com Spoto (2000), nos nós N_S da **SQL,** as *variáveis tabela* e as *variáveis host* podem estar tanto nos predicados das consultas (*queries*) quanto nas cláusulas dos comandos executáveis da **SQL**. Neste caso, uma referência a uma variável ***v*** foi definida como:

i) s-uso (uso na **SQL**) quando ***v*** afetar o resultado de uma consulta ou de uma computação em um comando executável da **SQL** (uso associado ao nó da **SQL**).

Um *s-uso* ocorre nos nós de N_S e pode afetar o fluxo nos arcos de saída dos nós de N_S quando houver ações de desvios para controle de erros (como *catch (SQLException <variável>)*). Essas ações afetam somente as *variáveis tabela* (SPOTO, 2000).

Em particular, o *uso* de uma *variável tabela* ocorre somente no nó *j* tal que $j \in N_S$ e *j* contiver um dos comandos da **SQL** responsáveis pela manipulação da base de dados (*SELECT, INSERT, UPDATE, DELETE*).

Como é comum a presença de comandos de tratamento de erros (que ocasionam desvios incondicionais a partir dos comandos executáveis da **SQL**) em ABDR, foi considerado que o *uso* da *variável tabela* está nos arcos de saída dos nós onde ocorre um *s-uso {(j, k)}*, aplicando a idéia de *t-uso* (uso persistente de ***t***) (SPOTO; JINO; MALDONADO, 1997).

Assim como definido em Spoto (2000) que aborda alguns tipos de fluxo de dados baseado em ABDR, define-se:

a) *Fluxo de dados* intra-classe: aplica-se ao teste de unidade (cada método) e ao teste de integração dos métodos de uma classe. Este tipo de fluxo é aplicado nas etapas de teste de cada classe da ABDR, iniciando a partir da etapa do teste de unidade e estendendo-se à etapa de teste de integração entre os métodos da classe. Cada método que pertence a classe é testado isoladamente observando os fluxos de dados das variáveis utilizadas na classe. A outra etapa de teste em que se aplica o fluxo de dados intra-classe é a do teste de integração baseado na dependência de dados persistentes;

b) *Fluxo de dados* inter-classe: aplica-se à integração das classes que compõem uma aplicação. Uma aplicação é formada por diversas classes que manipulam e consultam os dados existentes nas tabelas. As operações especificadas nos módulos de ABDR estabelecem fluxos de dados das tabelas para as classes e das classes para as tabelas que, indiretamente, estabelecem fluxos de dados entre as diferentes tabelas da aplicação. Dizemos que ocorre

fluxo de dados inter-classe quando um dado armazenado em uma tabela t_i por uma classe $Class_k$ é utilizado por outra classe $Class_j$, podendo ser em outro tempo de execução, sendo esse usado para interferir no armazenamento e, assim sucessivamente, gerando um fluxo de dados entre as classes através das variáveis persistentes (*variáveis tabela*). Também denotamos de Mod_i a classe que representa um Módulo de Programação.

Conforme a Figura 3.3, pode-se ter uma visão geral dos critérios de teste em ABDR, propostos por Spoto (2000).

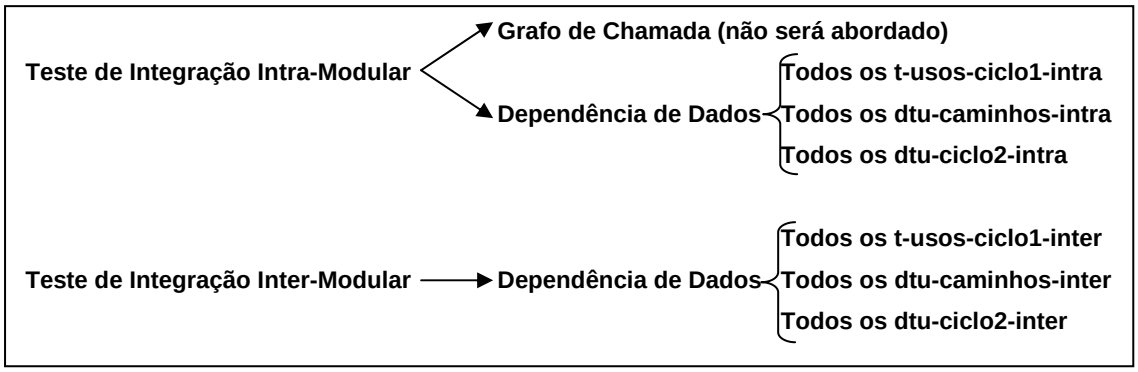


Figura 3.3 – Visualização dos critérios de teste em ABDR, propostos em (SPOTO, 2000)

Detalhes e exemplos destes fluxos serão expostos na Sessão 3.2.1 e 3.2.2.

3.1. Teste de Unidade de uma ABDR

No teste de um programa de ABDR, para exercitar os conjuntos de variáveis de Programa (P) ou *Host* (H) na etapa de teste de unidade pode ser realizada sob qualquer critério de teste estrutural (por exemplo, Família de Critérios de Fluxo de Controle (FCFD) ou Família de Critérios Potenciais Usos (FCPU)).

Considerando as *variáveis tabela* como *variáveis persistentes*, Spoto (2000) definiu os seguintes conjuntos:

(i) s-uso(j)={variáveis com *s*-uso no nó $j \in N_S$}

(ii) defT $<\ell,i>$ = {Variáveis ***t*** tal que cada ***t*** possui uma *definição persistente* pela *concatenação* da execução dos nós ℓe *i* da **SQL,** denotada por $<\ell, i>$ - no nó $\ell \in N_S$ existe um comando de manipulação (*INSERT*, *UPDATE* e *DELETE*) de ***t*** e no nó $i \in N_S$ existe um comando *COMMIT* - e os dois nós são sempre executados conjuntamente}.

(iii) dsu(v,i)={nós $j \in N_S$ tal que ***v*** $\in$ *s-uso(j)* e existe um caminho livre de definição *c.r.a.* ***v*** do nó *i* para o nó *j* e ***v*** pertence a *defg(i)* (onde *defg(i)* representa o conjunto de todas as variáveis com definição global do nó *i*)}.

Na Figura 3.4 é representado um grafo de um método com os principais comandos de manipulação da SQL (*INSERT*, *UPDATE*, *DELETE* e *SELECT*) para uma tabela ***t***. Este grafo foi construído através na Ferramenta JaBUTi (NARDI, 2005).

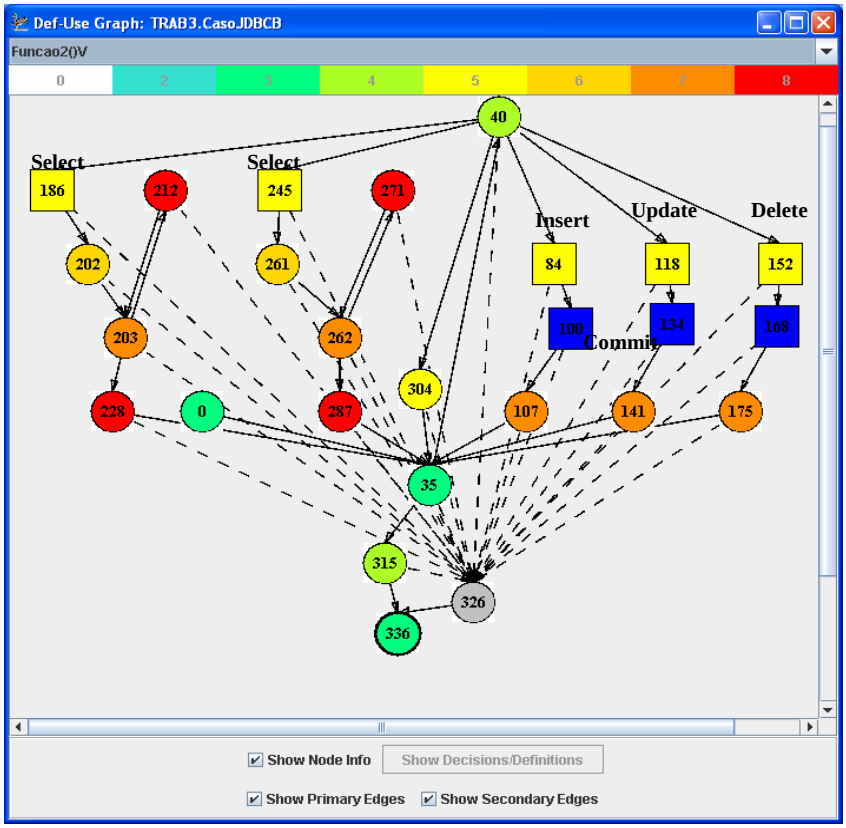


Figura 3.4: Função com os principais comandos de manipulação da SQL para uma tabela *t*

Baseado no trabalho de Spoto (2000) pode-se realizar as seguintes considerações a partir do grafo apresentado na Figura 3.4:

a) que existe a *definição persistente* em *variáveis tabela* ***t*** em um caminho se e somente se ambos os nós ℓ, $i \in N_S$ pertencem ao caminho. No exemplo da Figura 3.4 o caminho (40, 84, 326, 336) não contém definição persistente da variável tabela ***t*** visto que, apesar de o nó 84 conter o comando *INSERT*, o comando *COMMIT* não faz parte do caminho. Somente os caminhos contendo os pares de nós <84, 100>, <118, 134> ou <152, 168> contêm definição *persistente da variável* ***t*** (conjuntos *defT* <84, 100>, *defT*<118, 134>, *defT*<152, 168>). Em geral o nó de saída de um grafo não pertence ao conjunto N_S.;

b) Na ausência sintática do comando *COMMIT* em um procedimento que contém um comando de manipulação da SQL (*INSERT*, *DELETE*, *UPDATE*), considera-se que o comando *COMMIT* ocorre no próximo nó (por *default*), já comentado anteriormente. Se o grafo da Figura 3.4 não tivesse um nó com o comando *COMMIT*, as *definições persistentes* de ***t*** seriam, por exemplo: *defT*<84, 100>, *defT*<118, 134>, *defT*<152, 168>, que coincidentemente foram o mesmo nó do COMMIT na Figura 3.4.

Para os programas de aplicação, um caminho (n_1, n_2, ..., n_j, n_k) é um *du-caminho* c.r.a ***v*** se n_1 tiver uma definição global de ***v*** e:

✓ N_k tem um *c*-uso ou *s*-uso de ***v*** e (n_1, n_2, ..., n_j, n_k) é um caminho simples livre de definição c.r.a ***v***; ou

✓ O arco (n_j, n_k) tem um *p*-uso de ***v*** e (n_1, n_2, ..., n_j, n_k) é um caminho livre de definição c.r.a ***v*** e n_1, $n_{2, ..., } n_j$ é um caminho livre de laço.

As definições apresentadas neste item do trabalho (teste de unidade) podem ser estendidas para programas de aplicação no que se refere às *variáveis tabela*. Considere-se que toda variável tabela pode ter uma definição global em um nó *i*, isto é, *defg(i)*= {variável ***v***, tal que, ***v*** é uma variável do conjunto de todas as variáveis usadas em programas de aplicação}.

Considerando que as *variáveis tabela* possuem uma definição global no nó *i*, os critérios FCPU (Família de Critérios Potenciais Usos) ou da FCPD (Família de Critérios de Fluxos de Dados) podem ser aplicados no teste de unidade. As seguintes definições complementam os conceitos básicos para a definição dos critérios de teste de programas de ABDR, realizadas por (SPOTO, 2000):

(a) Associação *definição-s-uso* (*dsu*) é a tripla [*i, j,* ***v***] onde ***v*** $\in$ *defg(i)* e *j* $\in$ *dsu(v,i)*, válida para as variáveis ***H*** e ***T***, onde *defg(i)* = {variável ***v*** tal que ***v*** é definida no nó *i*}.

(b) Associação definição-t-uso é uma tripla [$<\ell,i>$, *(j, k),* ***t***] , onde: ***t*** $\in$ *defT(ℓ, i)*; *j* $\in$ *dsu(**t**, i)*; o arco *(j, k)* $\in$ $Arc_{out}(j)$ (*arco de saída de j*); os nós ℓ, *i* e *j* $\in N_S$; $<\ell, i>$ é a concatenação que estabelece uma definição *persistente* de ***t*** no nó *i* vinda do nó ℓ que alcança um uso de ***t*** no arco *(j,k)*; e existe um caminho *livre de definição* c.r.a ***t*** de *i* ao arco (*j, k*).

(c) dtu-caminho é um caminho livre de *definição persistente* (n_ℓ , .., n_i, ..., n_j, n_k) *c.r.a.* ***t*** dos nós $<n_\ell , n_i>$ até o nó n_k ou até o arco (n_j, n_k) onde ocorre um uso de ***t*** e o caminho (n_ℓ , .., n_i, ..., n_j, n_k) é um caminho livre de laço e nos nós n_ℓ e n_i ocorre uma *definição persistente* de ***t***.

Spoto (2000) definiu como conjunto factível (executável):

(d) fdsu(v, i) = {j $\in$ *dsu(v,i)* tal que a associação *[i, j,* ***v****]* , *j* $\in N_S$, é executável};

Em Spoto (2000) foram propostos os seguintes critérios aplicados ao teste de unidade, considerando G(UP) um grafo de programa e Π um conjunto de caminhos completos de G(UP). Seja Γ um conjunto de *tuplas* $\{\tau_1, \tau_2, ..., \tau_c\}$ pertencentes a uma dada tabela **t**:

a) ***Todos os t-usos***: requer para todas as associações *definição-t-uso* [$<\ell, i>$, (*j,k*), ***t***] que pelo menos um caminho $\pi = (n_{in}, ..., \ell, ...,i, ...,j,k,...,n_{out}) \in \Pi$, livre de *definição persistente* de $<\ell,i>$ até (*j,k*) c.r.a ***t***, seja exercitado pelo menos uma vez para a mesma *tupla* $\tau \in \Pi$;

b) ***Todos os dtu-caminhos***: requer que todos os *dtu-caminhos* (n_ℓ,..., n_i, ..., n_j, n_k) sejam executados pelo menos uma vez. A associação deve ser exercitada para a mesma *tupla* $\tau \in \Gamma$.

Segundo Spoto (2000), um conjunto de caso de teste **Tc**, que resulta em Π e Γ:

- Satisfaz o critério (Todos os *t-usos*) se, para todo par $<\ell,i>$ de nós $\in$ G(UP) com *defT* $<\ell,i> \neq \phi$ para toda variável $\boldsymbol{t} \in$ *defT* $<\ell,i>$, Π incluir todas as associações [$<\ell,i>$, *(j,k),* $\boldsymbol{t}$] onde $<\ell,i>$ contém uma *defT* $<\ell,i>$ e existir um t-uso em (*j*,*k*) sendo $j \in$ *fdsu* ($\boldsymbol{t}$,*i)* e $j \in N_s$; e existe um caminho livre de definição persistente de $<\ell,i>$ até (*j*,*k*) . A associação será satisfeita se e somente se for exercitada para a mesma *tupla* τ.

- Satisfaz o critério (Todos os *dtu-usos*) se, para todo par $<\ell,i>$ de nós $\in$ G(UP) com *defT* $<\ell,i> \neq \phi$, Π incluir todos os dtu-caminhos de $<\ell,i>$ até *(j,k)* c.r.a cada variável $\boldsymbol{t} \in$ *defT* $<\ell,i>$ para todas as associações [$<\ell,i>$, *(j,k),* $\boldsymbol{t}$] tal que $<\ell,i>$ contém uma *defT* $<\ell,i>$ e existir um t-uso em (*j*,*k*) sendo $j \in$ *fdsu* ($\boldsymbol{t}$,*i)* e $j \in N_s$; e existe um caminho livre de definição persistente de $<\ell,i>$ até (*j*,*k*) . A associação será satisfeita se e somente se for exercitada para a mesma *tupla* τ.

Estes critérios têm como objetivo requerer que todo sub-caminho que inicia no nó ℓ da SQL (*INSERT* ou *DELETE* ou *UPDATE*) e passa pelo nó *i* (*COMMIT*), onde ocorre a definição persistente de $\boldsymbol{t}$, e alcança um arco de saída do nó da SQL, onde ocorre o uso de $\boldsymbol{t}$ (*t-uso*) com comandos (*INSERT*, *DELETE*, *UPDATE* ou *SELECT*), sejam executados pelo menos uma vez para a mesma *tupla* τ.

Para Spoto (2000), os dois critérios acima foram criados ao considerar as *variáveis de tabela* tratadas como *variáveis persistentes*. Os elementos requeridos pelos critérios *todos-t-usos* e *todos-dtu-caminhos* só são satisfeitos se forem executados para a mesma *tupla*. A exigência de mesma *tupla* é fundamental para o teste de *variáveis persistentes*.

3.2. Teste de Integração em uma ABDR

O teste de integração é aplicado depois que todas as unidades que compõem um programa foram devidamente testadas isoladamente (teste de unidade). O teste de integração pode ser visto como uma técnica sistemática para a construção da estrutura do software procurando revelar defeitos associados à interação entre as *Unidades do Programa (UP)* ou entre os *Módulos de Programa (Mod).*

A dependência dos dados é um aspecto fundamental associado à integração entre unidades de programa e módulos de programas. As dependências de dados existentes entre as UPs de programas convencionais são ocasionadas pelas variáveis globais ou por variáveis passadas por parâmetros através de comandos de chamadas. Os programas de aplicação possuem dependência de dados baseadas nos comandos de chamada, como acontece nos programas convencionais; e **dependências de dados baseadas nas tabelas da base de dados**. As dependências de dados entre as UPs e os Módulos de Programas de uma ABDR dão ensejo a novas abordagens de integração (SPOTO, 2005).

A abordagem adotada em Spoto (2005), sobre teste estrutural para programas de ABDR considera dois tipos de fluxo de dados para as variáveis tabela: fluxo de dados intra-modular (aborda o teste de unidade e teste de integração intra-modular) e fluxo de dados inter-modular; em ambos os casos, a análise do fluxo de dados é usada para abstrair requisitos para o teste de integração *intra-modular* e *inter-modular* (lembrando que iremos chamar de intra-classe e inter-classe respectivamente). O fluxo de dados *intra-classe* é observado entre os métodos de uma mesma classe; o fluxo de dados *inter-classe* ocorre entre métodos de classes distintas.

Veremos a seguir os critérios de teste aplicados ao Teste de Integração Intra-Classe e Teste de Integração Inter-Classe.

3.2.1. Critérios de Teste de Integração Intra-Classe (Intra-Modular)

Os critérios de teste de integração intra-classe visam a exercitar as associações *definição-t-uso* determinadas pelos comandos de manipulação da base de dados. São divididos segundo duas abordagens de teste: **baseada no grafo de chamada** e **baseada na dependência dos dados**. Tanto os critérios de integração baseados no grafo de chamadas como os critérios baseados na dependência de dados determinada pelas tabelas da base de dados, visam a exercitar as associações referentes às *variáveis de programas* (P) ou *variáveis tabela* (T), de acordo com o enfoque estabelecido pelos critérios analisados (SPOTO, 2000).

Os critérios de teste propostos em Spoto (2000), para associar as *variáveis persistentes* devem ser satisfeitos com a mesma *tupla*, forçando o testador a gerar casos de testes específicos para exercitá-la. Deste modo, o teste deve ser executado de maneira controlada, não sendo suficiente usar qualquer *tupla* de ***t***. Por outro lado, com a exigência do uso da mesma *tupla* para satisfazer uma associação *definição-t-uso*, pode aumentar o número de elementos requeridos não factíveis (isto é, não executáveis com a mesma *tupla*).

3.2.1.1. Critérios de Teste de Integração Baseados no Grafo de Chamadas

Os critérios definidos neste item são baseados na mesma idéia dos critérios Potenciais Uso de Integração definidos e analisados em Vilela (1998). O grafo de chamada é um multi-grafo, podendo haver mais de uma chamada entre duas *Unidades de Programa*, o que corresponde à ocorrência de mais de um arco ligando os respectivos nós do grafo. Essa abordagem considera a integração das *Unidades de Programas* (UPs), feita *duas-a-duas* ("*parwhise*") e os requisitos de teste são conseqüentemente derivados para cada par de UPs.

Spoto (2000) estendeu os critérios de integração propostos por Vilela (1998), definindo critérios que têm como finalidade exercitar as associações envolvendo as *variáveis tabela* ***t***. Porém, esses critérios não serão abordados neste trabalho.

Os critérios de integração são complementares aos de teste de unidade e visam a exercitar classes de erros distintas.

3.2.1.2. Critérios de Teste de Integração Baseados na Dependência dos Dados

A principal característica de programas de aplicação, que distingue-os dos programas convencionais, conforme Spoto (2005), é a persistência dos dados da base de dados. As unidades de programas (ou métodos no caso de Programas em Java) e módulos de programa (ou classes em casos de Java), se autorizados, podem ter acesso a esses dados a qualquer momento. A persistência é uma propriedade que não existe exclusivamente em Banco de Dados Relacional, podendo existir para outros programas com características semelhantes (SPOTO, 2005).

Em ABDR, a mesma tabela da base de dados pode estar disponível para diferentes classes e, conseqüentemente, para diferentes métodos de uma mesma classe ou classes diferentes. Neste caso, duas unidades de programa (métodos) podem possuir uma dependência de dados entre elas, mesmo não exigindo chamadas entre as unidades.

Os critérios de Teste de Integração Baseados na Dependência dos dados, segundo Spoto (2000) baseiam-se nas dependências de dados existentes entre os *procedimentos* (métodos) de um mesmo *Módulo de Programa (classe)* em relação a uma *variável tabela* da base de dados e requerem a mesma *tupla* para satisfazer *a associação definição-t-uso*. Para os programas em Java denominaremos de módulo de programa (*Mod*) para Classe e de unidade de programa (*UP*) para representar o método da classe. Quando uma unidade de programa UP_A tiver uma definição persistente da variável ***t*** e em outra unidade UP_B tiver um *uso* de ***t*** (*t*-

uso); os pares (UP_A, UP_B) serão requeridos pelo critério com um ciclo de dependência de dados (denomina-se *ciclo 1*). A execução dessa associação poderá ser efetuada em apenas uma execução de cada unidade.

Em Spoto (2005), é explicado que duas unidades de programa UP_A e UP_B, não necessariamente distintas, possuem uma dependência de dados de UP_A para UP_B com relação à variável tabela ***t*** (isto é, altera o conteúdo da tabela, cujo estado passa de e_i para e_{i+1}) e a unidade de programa UP_B possua um ponto no programa que usa a variável tabela ***t*** (com o estado de ***t*** igual a e_{i+1}). Uma variável tabela ***t*** com uma definição persistente em uma unidade UP_A (passando a variável ***t*** para um estado e_A) estará em um estado consistente (representado por e_A) se e somente se não existir redefinição persistente de ***t*** em pelo menos um subprograma UP_A no nó de saída. Existe uma dependência de dados c.r.a variável tabela ***t*** se ***t*** tiver uma *definição persistente* em uma unidade UP_A e existir um caminho livre de definição c.r.a ***t*** incluindo o sub-caminho de UP_A a partir da definição persistente até o nó de saída e um sub-caminho do nó de entrada de uma UP_B até o arco (n_j, n_k) que contém um ***t*-*uso***.

Da mesma forma definida em (SPOTO 2005) considerou-se dois tipos de dependências de dados c.r.a ***t***:

- **Dependência interna ou intra-classe**: ocorre quando existir uma dependência de dados entre métodos de uma mesma classe, com relação a uma ou mais tabelas, mesmo quando não existir um ponto de chamada entre elas;

- **Dependência externa ou inter-classe**: ocorre quando existir uma dependência de dados entre dois ou mais métodos de classes diferentes c.r.a ***t***. Isto é, ***t*** deve estar disponível para as duas classes.

Considerando que $G(UP_A)$ e $G(UP_B)$ sejam os grafos dos métodos A e B, respectivamente, de uma mesma classe e que são envolvidas na integração e $\prod$ seja o conjunto dos caminhos completos em $G(UP_A)$ e $G(UP_B)$ e Γ o conjunto de *tuplas* $(\pi_a, \pi_b) \in \prod$

implicando na existência da concatenação π_a, π_b ,onde $\pi_a \in G(UP_A)$ e $\pi_b \in G(UP_B)$, (SPOTO, 2000) resultando em três critérios de teste, que para este trabalho será adaptado para programas em Java:

a) ***Todos os t-usos-ciclo1-intra***: $\prod$ e Γ satisfazem o critério (*todos os t-usos-ciclo1-intra*) para o par de unidades (métodos) UP_A e UP_B pertencentes à mesma classe se, para todos os pares de caminhos $(\pi_a, \pi_b) \in \prod$, π_a inclui o par de nós $<\ell,i> \in G(UP_A)$ tal que $defT<\ell,i> \neq \phi$ c.r.a ***t***, e existir um *caminho livre de definição* c.r.a ***t*** de $<\ell,i>$ até n_{out} de $G(UP_A)$; e π_b inclui o arco $(j, k) \in G(UP_B)$ onde $j \in fdsu(\mathbf{t},i)$ e existe um ***t-uso*** em (j,k) e existir um *caminho livre de definição persistente* c.r.a **t** de n_{in} até o arco (j, k); a associação é satisfeita se for exercitada através da mesma *tupla* τ.;

b) ***Todos os dtu-caminhos-intra***: $\prod$ e Γ satisfazem o critério (*todos os dtu-caminhos-intra*) para o par de unidades (métodos) UP_A e UP_B pertencentes à mesma classe se, para todos os pares de caminhos $(\pi_a, \pi_b) \in \prod$, π_a incluir um *dtu-caminho* do nó ℓ até o nó n_{out} de $G(UP_A)$ passando pelo par de nós $<\ell,i> \subseteq G(UP_A)$ onde $defT<\ell,i> \neq \phi$ c.r.a ***t***, através da *tupla* τ e π_b incluir um *dtu-caminho* c.r.a ***t*** do n_{in} de $G(UP_B)$ até o arco $(j, k) \in G(UP_B)$ sendo $j \in fdsu(\mathbf{t}, i)$ e existe um ***t-uso*** em (j,k) através da mesma *tupla* τ e existir um *caminho livre de definição persistente* que vai do par $<\ell,i>$ em UP_A até o arco (j, k) em UP_B c.r.a ***t*** na composição dos caminhos π_a, π_b.

Para exemplificar os critérios propostos em Spoto (2005) pode-se verificar na Figura 3.5, a existência dos métodos *inserirDados()* e *verificarDados()* ambos pertencentes à classe Banco.Java. O método *inserirDados()* define a variável tabela **Cliente** inserindo a *tupla* cujo *cpf=2* nos pares de nós *defT<62, 78>*. O método *verificarDados()* faz um uso ao selecionar a mesma *tupla* com *cpf=2*, no arco (62, 69). Lembrando que ocorre na mesma *tupla* da tabela. Também é possível observar na Figura 3.5, que o caminho $\prod_a$ 0, 7, 26, **62**, 69, **78**, 86, 111 e

142, no método *inserirDados()*, passa pelo par de nós <*62, 78*> onde existe uma definição persistente em defT<62, 78>, na qual pode-se definir a variável tabela Cliente inserindo a *tupla* cujo cpf = 2 e em seguida efetuar um *t-uso persistente* selecionando a *tupla* com *cpf = 2*, no método *verificarDados()*, exercitando o caminho $\prod_b$ 0, 7, **26, 62**, 69 passando pelo arco (26, 62) onde ocorre um *t-uso*. Como uma estratégia de implementação ambos dtu-caminhos (*definição t-uso*) de uma mesma tabela devem ser executadas pela mesma *tupla*.

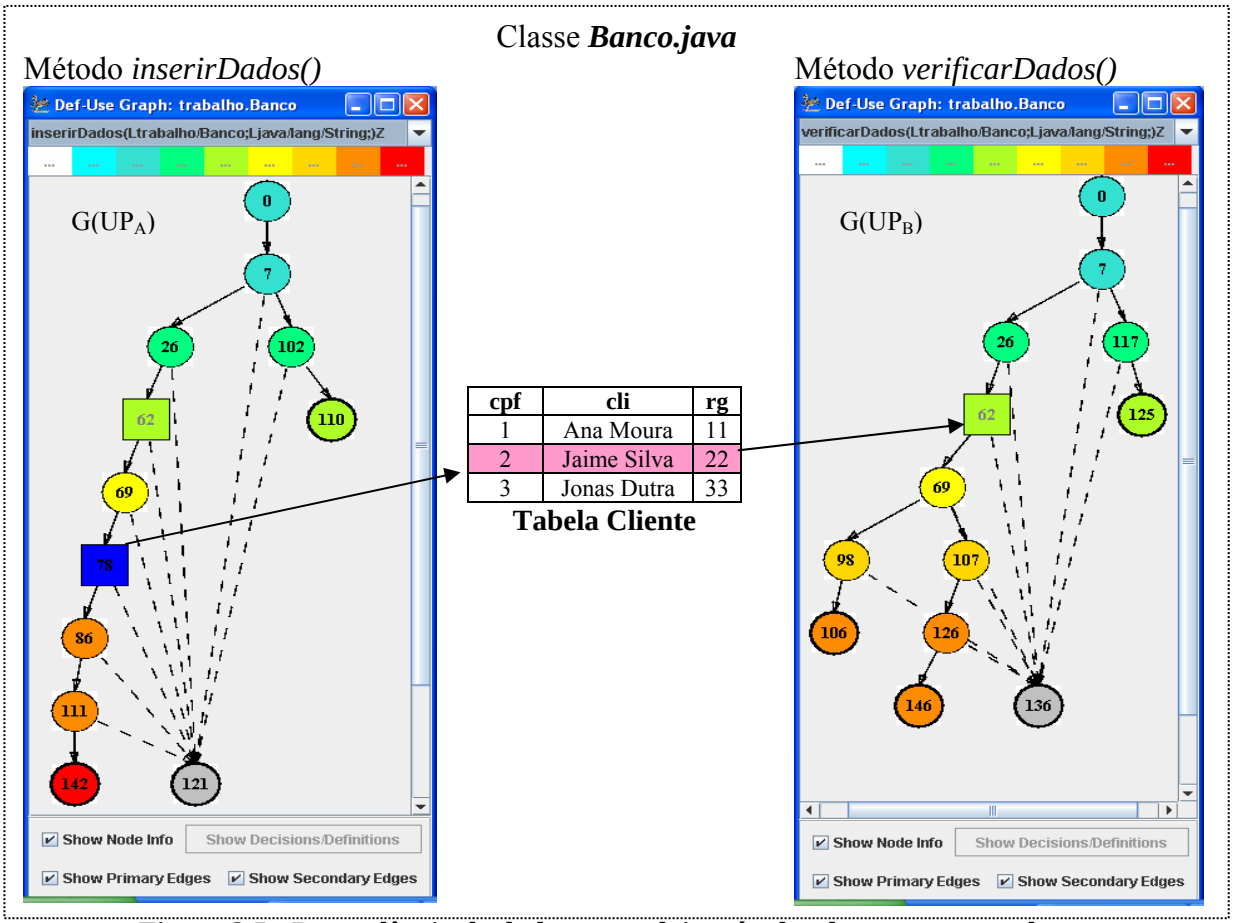


Figura 3.5 - Dependência de dados entre dois métodos de uma mesma classe.

Existem dependências que não podem ser exercitadas com a execução de apenas duas unidades (uma de *definição* e outra de *uso*), mas necessitam da execução de outra unidade para satisfazer a associação. Isso ocorre quando existem dependências múltiplas, ou seja, para definir a variável ***t*** é necessário definir a variável ***t'*** e, para isso, pode ser necessário

executar a unidade que define a variável ***t'*** para depois executar a unidade que define ***t*** e finalmente executarmos a UP que usa ***t***.

Observa-se que podem existir dependências múltiplas exigidas em um *t-uso* obrigando assim em mais de dois ciclos de execução. Considerando três unidades: UP_1, UP_2 e UP_3 e supondo que a Unidade UP_3 possua *t-uso* de duas variáveis tabela que são definidas persistentemente em UP_1 e UP_2 necessitando que sejam executadas as Unidades UP_1, UP_2 e UP_3 na seqüência. Outro caso ocorre quando a variável tabela a ser definida na unidade UP_2 necessite de uma outra variável definida em UP_1 (Exemplo: para que uma venda seja definida, é necessária a definição da tabela cliente) obrigando assim a seqüência de execução UP_1, UP_2 e UP_3.

c) ***Todos os t-usos-ciclo2-intra***: $\prod$ e Γ satisfazem o critério (*todos os t-usos-ciclo2-intra*) para o par de unidades (métodos) UP_A, UP_B e UP_C pertencentes a mesma classe se, para todos os pares de caminhos $(\pi_a, \pi_b, \pi_c) \in \prod$, π_a é o caminho que contém um par de nós $<\ell', i'>$ que *define* a variável ***t'*** pela *tupla* τ'; o caminho π_b contém um par de nós $<\ell, i>$ que define ***t*** pela *tupla* τ podendo ou não ter uma dependência de ***t'***, e o caminho π_c possuir um nó que contém um *uso* de ***t*** ou ***t'*** pela mesma *tupla* τ e/ou τ', respectivamente, e o caminho π_a passar pelos nós $<\ell', i'> \in G(UP_A)$ com $defT<\ell', i'> \neq \phi$ c.r.a ***t'***; e existir um *caminho livre de definição* c.r.a ***t'*** de *i* até n_{out} de $G(UP_A)$; o caminho π_a passar pelos nós $<\ell, i> \in G(UP_B)$ onde $defT<\ell, i> \neq \phi$ c.r.a ***t***; existir um *caminho livre de definição* c.r.a ***t*** de *i* até n_{out} de $G(UP_B)$; o caminho π_b passar pelo arco $(j, k) \in G(UP_C)$ existe um ***t-uso*** em (j, k) e $j \in fdsu(\mathbf{t}, i)$ (sendo **t** e/ou **t'**) e existir um *caminho livre de definição* c.r.a ***t*** e/ou ***t'*** de n_{in} até o arco (j, k). A associação $[<\ell', i'>, UP_A, <\ell, i> UP_B, (j, k), UP_C, \{\mathbf{t'}, \mathbf{t}\}]$ é satisfeita se e somente se a mesma *tupla* τ (τ') usada para satisfazer a definição de ***t*** (***t'***) e também usada para satisfazer o *uso*.

Para exemplificar, utilizou-se a Figura 3.6, (usando-se apenas um grafo da $UP_{inserirDados,}$ porém, com duas execuções simultâneas sendo que a primeira execução para a variável tabela Fornecedor e na segunda para a variável tabela Produto) cujo caminho 0, 7, 26, 62, 69, 78, 86, 111 e 142, no método *inserirDados()* passa pelo par de nós <*62, 78*> onde existe uma defT<62, 78>, na qual pode-se definir uma variável tabela Fornecedor inserindo a *tupla* cujo *cnpj = 333* em seguida executa-se método *inserirDados()* na *tupla* cujo *cod_prod = 3* na variável tabela Produto, cujo fornecedor é o fornecedor com *cnpj = 333*. Após a definição persistente da variável tabela Produto inicia-se a execução do método *verificarDados()* que selecionam as *tuplas* com *cod_prod = 3* e *cnpj = 333* nas variáveis tabelas Produto e Fornecedor respectivamente.

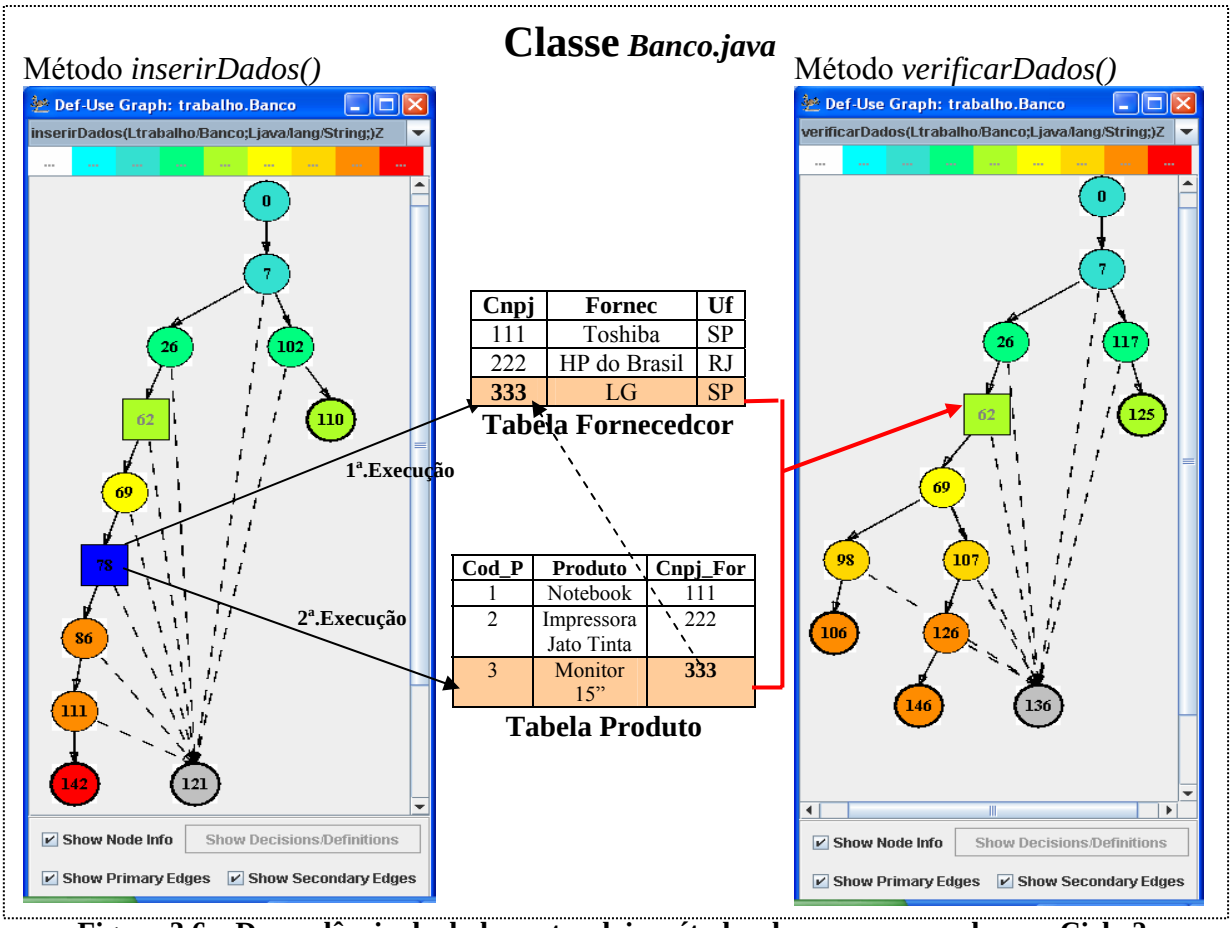


Figura 3.6 – Dependência de dados entre dois métodos de uma mesma classe – Ciclo 2.

Dois enfoques apresentados por Spoto (2000) podem ser considerados:

a) Utilizar as mesmas *tuplas* para satisfazer os pares definição-uso na integração; ou

b) Utilizar *tuplas* diferentes para satisfazer os elementos requeridos não cobertos com a mesma *tupla*.

O primeiro enfoque é mais conservador e mostrou, na execução de um exemplo de aplicação em Spoto (2000), ser melhor para detecção de defeitos do que o segundo.

Como exemplo, será utilizado um sistema em Linguagem Java, onde uma das classes que manipulam os dados é a Banco.Java, que contém os métodos (UPs) *inserirDados*(*bd*, *query*), *alterarDados*(*bd*, *query*), *removerDados*(*bd*, *query*) e *verificarDados*(*bd*, *query*).

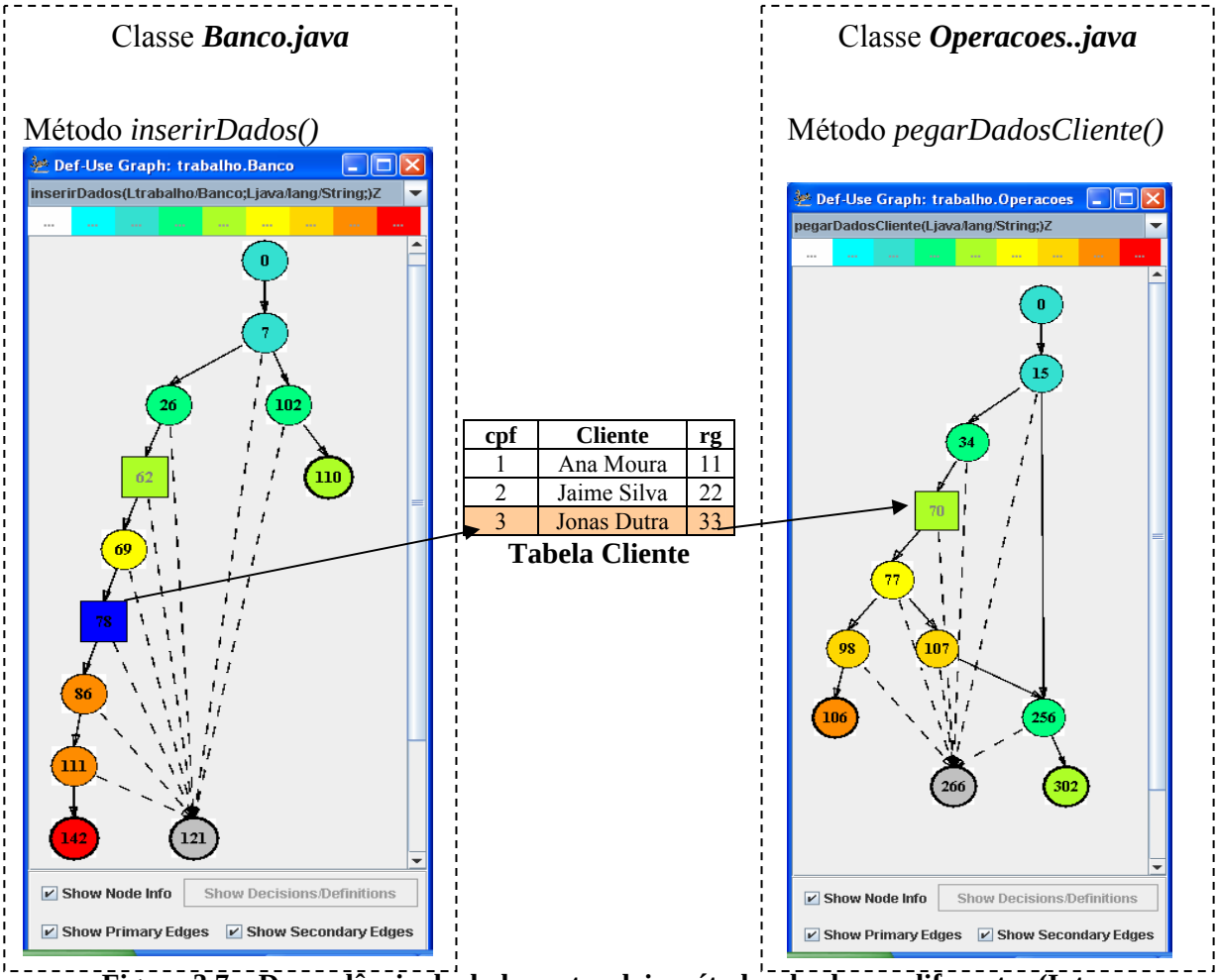
3.2.2. Critérios Aplicados ao Teste de Integração Inter-Classe (Inter-Modular)

Os critérios de integração inter-classe, são semelhantes aos critérios de integração intra-classe, apenas com a diferença de que os métodos associados devem pertencer a classes distintas, visando exercitar as associações de variáveis de tabela que são definidas em um método de uma classe α e são usadas em métodos de outra classe β, possuem também, três tipos. Considere dois métodos UP_A, pertencente a uma classe Mod_x , e o método UP_B, pertencente a uma outra classe Mod_y. Considere também os mesmos conjuntos Π e Γ definidos anteriormente. Temos então:

a) ***Todos os t-usos-ciclo1-inter***: este critério é idêntico ao critério *todos os t-usos-ciclo1-intra*, com a única diferença que os métodos UP_A e UP_B pertencem a classes distintas;

b) ***Todos os dtu-caminhos-inter***: este critério é idêntico ao critério *todos os dtu-caminhos-intra*, com a única diferença que o método onde existe uma definição persistente (UP_A) pertence a uma classe e o método onde existe um ***t-uso*** pertence a outra classe;

c) ***Todos os t-usos-ciclo2-inter***: este critério é idêntico ao critério *todos os todos os t-usos-ciclo2-intra,* com a diferença que os métodos *associados* devem pertencer classes distintas.



cpf	Cliente	rg
1	Ana Moura	11
2	Jaime Silva	22
3	Jonas Dutra	33

Tabela Cliente

Figura 3.7 – Dependência de dados entre dois métodos de classes diferentes (Inter-Classe).

Para exemplificar os critérios propostos em Spoto (2005) pode-se verificar na Figura 3.7, a existência dos métodos *inserirDados()*, pertencente à classe Banco.java e o método *pegarDadosCliente()*, pertencente à classe Operacoes.java. O método *inserirDados()* define a variável tabela **Cliente** inserindo a *tupla* cujo *cpf=3* nos pares de nós *defT<62, 78>*. O método *pegarDadosCliente()* faz um uso ao selecionar a mesma *tupla* com *cpf=3*, no arco (70, 77), na mesma *tupla* que foi definida. Também é possível observar na Figura 3.7, que o

caminho 0, 7, 26, **62**, 69, **78**, 86, 111 e 142, no método *inserirDados()*, passa pelo par de nós *<62, 78>* onde existe uma definição persistente em defT<62, 78>, na qual pode-se definir a variável tabela Cliente inserindo a *tupla* cujo cpf = 3 e em seguida efetuar um *t-uso persistente* selecionando a *tupla* com *cpf = 3*, no método *pegarDadosCliente()*, exercitando o caminho 0, 15, 34, **70**, **77** passando pelo arco (**70, 77**) onde ocorre um *t-uso*. Como uma estratégia de implementação ambos dtu-caminhos (*definição t-uso*) de uma mesma tabela devem ser executadas pela mesma *tupla.*

3.3. A Eficácia do Teste na detecção de erros

No Dicionário Aurélio básico da língua portuguesa (FERREIRA, 1988), temos que eficácia é o que produz o efeito desejado; que dá bom resultado ou o que age com eficiência.

Segundo Bio (1985), **eficácia** diz respeito a resultados, a produtos decorrentes de uma atividade qualquer. Trata-se da escolha da solução certa para determinado problema ou necessidade. A eficácia é definida pela relação entre resultados pretendidos/resultados obtidos.

A **eficácia de um critério de teste** está relacionada à habilidade do critério em levar o testador a selecionar dados que tenham uma boa chance de revelar defeitos do programa ainda não revelados (BATISTA, 2003).

Devido à diversidade de critérios de testes existentes surge a questão de qual critério utilizar para se obter o melhor resultado com o menor custo. Mesmo utilizando técnicas e critérios existentes, dividindo a atividade de teste em várias fases e utilizando as ferramentas de teste, não se pode garantir um software livre de erros (VINCENZI, 1998).

Segundo Vincenzi (1998), vários critérios e técnicas de teste têm sido elaborados visando a fornecer uma maneira sistemática e rigorosa para selecionar um subconjunto de

domínio de entrada e ainda assim, ser eficaz para revelar a presença de erros existentes respeitando as restrições de tempo e custo associados a um projeto de software.

A principal característica do programa de ABDR com comandos SQL é que os valores das relações da base de dados podem ser manipulados por diferentes usuários, controlados pelo SGBDR que dispõe de recursos próprios para garantir a segurança e estabilidade de transação dos dados armazenados nas relações e como qualquer programa, podem possuir códigos mal formulados durante a implementação ou até consultas mal formuladas derivando dados incorretos aos usuários (SPOTO, 2000).

Além da característica descrita acima, tem-se também o aspecto da persistência dos dados na base de dados, da dependência de dados baseadas nas tabelas da base de dados, ocasionadas entre os métodos de uma mesma classe (dependências intra-modular) e ocasionada entre os métodos de classes diferentes (dependências inter-modular).

Em geral, nos testes de programas convencionais, a escolha dos dados de testes baseado nos elementos requeridos dos critérios de fluxo de dados (FCFD e FCPU) são focados nas definições e usos das variáveis cujos valores são armazenados em memória principal. Em testes de ABDR, os dados de testes baseados nos elementos requeridos pelos critérios que exercitam definições e usos das variáveis tabela, são focados nas características de armazenamento persistente. Sendo assim, o foco de abrangência passa a ser mais rígido e exige mais planejamento e acompanhamento dos resultados voltados para os esquemas de banco de dados.

Os critérios estabelecidos para avaliar ABDR com a abordagem de variáveis persistentes vêm complementar as análises de fluxo de dados ainda não exercitadas com os critérios cujas abordagens eram apenas variáveis de armazenamento de variáveis em memória principal.

No estudo de caso deste trabalho, o sistema avaliado utiliza uma classe para tratar as principais manipulações de banco de dados (*INSERT, UPDATE E DELETE*), sendo que as SQLs são submetidas ao parâmetro e também utiliza outra classe para manipulações de consultas. Os comandos SQLs são gerados em uma terceira classe, cujas operações (*INSERT, DELETE, UPDATE, SELECT*) e tabelas são passadas por parâmetros. Após a geração do comando da SQL é enviado à classe responsável por executar efetivamente a ação desejada.

Podem existir situações em que os métodos de uma mesma classe possuam apenas comandos que caracterizam ocorrências de uso de tabelas, não sendo possível a geração da integração intra-classe (intra-modular) para exercitar as dependências existentes nos métodos desta classe (classes que só executam relatórios). Nesse caso, a integração inter-classe (inter-modular) complementa a etapa de teste exercitando tais métodos cujos comandos possuem apenas *Queries,* integrando, assim, com classes que possuem métodos com definição persistente. Pode-se verificar tal integração na Figura 3.8, onde na classe Banco.Java existem os métodos na cor azul (*INSERT, UPDATE, DELETE*) que definem ou usam uma variável tabela e os demais métodos na cor laranja, tanto na classe *Banco.Java*, quanto na classe *Operações.Java* que apenas exercitam o uso de uma variável tabela (*SELECT*). Sem a integração inter-classe, não seria possível detectar nenhum tipo de erro proveniente do fluxo de dados entre tabelas e programas que possuem somente uso.

Os critérios de teste estrutural em ABDR, estudados neste trabalho, contribuem assim para melhorar as chances de detecção de dados de teste para variáveis persistentes.

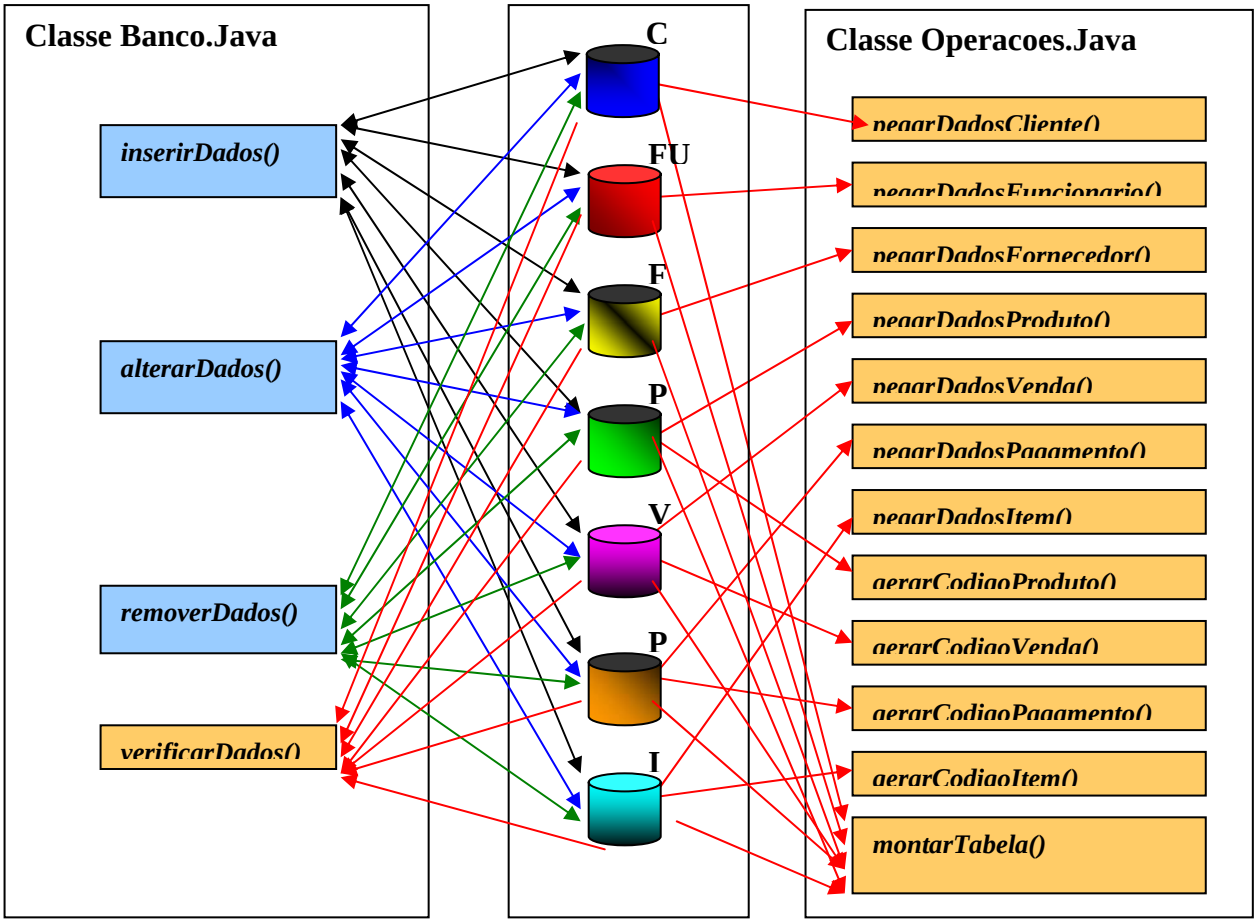


Figura 3.8 – Dependência Inter-Classe

3.3.1. Plano de Teste

A Norma IEEE 829 (IEEE, 1998) descreve um conjunto de documentos para as atividades de teste de um produto de software. Os oito documentos definidos pela norma, que cobrem as tarefas de planejamento, especificação e relato de testes, são: plano de teste, especificação de projeto de teste, especificação de caso de teste, especificação de procedimento de teste, diário de teste, relatório de incidente de teste, relatório-resumo de teste e relatório de encaminhamento de item de teste. A norma separa as atividades de teste em três etapas: preparação do teste, execução do teste e registro do teste. Embora a norma possa ser utilizada para o teste de produtos de software de qualquer tamanho ou complexidade, projetos pequenos ou de baixa complexidade podem agrupar alguns documentos propostos, diminuindo o gerenciamento e os custos de produção dos documentos. Além disso, o

conteúdo dos documentos também pode ser abreviado. Mais do que apresentar um conjunto de documentos que devem ser utilizados ou adaptados para determinadas empresas ou projetos, a norma apresenta um conjunto de informações necessárias para o teste de produtos de software. Sua correta utilização auxiliará a gerência a se concentrar tanto com as fases de planejamento e projeto quanto com a fase de realização de testes propriamente dita; isso evitaria a perigosa armadilha de só começar a pensar no teste de um produto de software após a conclusão da fase de codificação (CRESPO, 2004).

Segundo Pressman (1995), uma estratégia de teste de software integra técnicas de projeto de casos e de teste numa série bem definida de passos que resultam na construção bem sucedida do software. Esses passos podem envolver uma combinação das técnicas e aplicação de diferentes critérios de teste. Geralmente, a escolha de um critério de teste é guiada pelos fatores eficácia (número de defeitos revelados), e custo (número de casos de teste requeridos).

Para se desenvolver uma atividade de teste de maneira a atingir o objetivo de qualidade que se espera, não existe um teste único a ser realizado e sim uma combinação de vários testes ao longo do projeto. A estratégia descrita por Pressman (1995) prevê as etapas de teste de unidade, teste de integração, teste de validação e teste de sistema, abordados no Capítulo 2.

Geralmente, segundo Pressman (1995), essa atividade de teste envolve algumas fases tais como:

a) Planejamento do Teste: produz o plano de teste. O plano de teste definirá as etapas e categorias de teste a serem conduzidos, e inclui uma lista de: requisitos a serem testados ou verificados; critério de aceitação, aprovados pelo responsável do negócio (incluindo considerações de desempenho); regras e responsabilidades; ferramentas e técnicas a serem utilizadas e um cronograma para o teste;

b) Projeto de Caso de Teste: transforma os requisitos do sistema e comportamentos esperados pela aplicação, como especificados, em casos de teste documentados;

c) Desenvolvimento do Teste: transforma os casos de teste em programas, *scripts*, que exercitarão o sistema em desenvolvimento - ferramentas próprias devem ser usadas;

d) Execução do Teste e análise dos resultados: executam propriamente os *scripts* que irão gerar resultados para serem confrontados com os valores de entrada.

Estudos empíricos são motivados pela necessidade de se saber avaliar as tecnologias que surgem a cada momento para construir uma base sólida de informações a fim de contribuir para a evolução dessas tecnologias e da área de computação como um todo. Avaliar o custo/beneficio das técnicas de Teste de Software tem sido uma preocupação constante dos pesquisadores nas últimas décadas (BARBOSA, VINCENZI & MALDONADO, 1998; BASILI & REITER, 1981).

Em Werner (2005), tem-se que pesquisas em estudos empíricos procuram, por meio da comparação entre os critérios, obter uma estratégia que seja eficaz para revelar a presença de erros no programa, ao mesmo tempo em que apresente um baixo custo de aplicação.

Pesquisa de estudo de caso é um método observacional, ou seja, é feito pela observação de uma atividade ou de um projeto em andamento (WERNER, 2005).

Para pesquisas empíricas da comunidade de computador de IEEE, Kitchenham et al. (2002), descrevem alguns *guidelines* (diretrizes) de pesquisa para melhorar a pesquisa e relatórios de processos. Estes propõem um conjunto preliminar dos *guidelines* que são baseados em uma revisão dos *guidelines* da pesquisa desenvolvidos para pesquisadores médicos e em suas experiências em fazer e revisar pesquisas de Engenharia de Software. A finalidade destes pesquisadores é ajudar a revisores, pesquisadores e analistas conduzindo e avaliando estudos empíricos, além de evitar as principais armadilhas em suas atividades de pesquisa e informar corretamente sua pesquisa. Os autores acreditam que a adoção de tais

guidelines melhora a qualidade de estudos individuais, além de aumentar a probabilidade de poder utilizar uma meta-análise para combinar os resultados de estudos relacionados. Porém acreditam que existe uma necessidade de ter um debate mais abrangente antes da comunidade de pesquisa de Engenharia de Software desenvolver e concordar em *guidelines* definitivos.

Kitchenham et al. (2002), consideraram guidelines para o que fazer e o que não fazer sob seis tópicos:

1. **Contexto experimental**: assegurar-se de que os objetivos da pesquisa estejam definidos corretamente e assegurar-se de que a descrição da pesquisa forneça bastante detalhe para outros pesquisadores;
2. **Projeto experimental**: descreve os produtos, os recursos e os processos envolvidos no estudo, incluindo: a) a população que está sendo estudada; b) a razão e a técnica para provar essa população; c) o processo para alocar e administrar os tratamentos e d) os métodos usados para reduzir influências e determinar o tamanho da amostra;
3. **Administração do experimento e levantamento de dados**: envolve coletar as medidas experimentais do resultado. Sendo este um problema particular para experiências do software porque essas medidas não são padronizadas. Assim o objetivo *guidelines* desta fase, é assegurar que o processo de levantamento de dados foi bem definido, o bastante para que uma experiência possa ser replicada. Nesta fase também inclui monitorar o gravar todos os desvios do experimento como: desistentes e perguntas sem respostas.
4. **Análise**: os *guidelines* da análise, apontam em assegurar que os resultados experimentais estejam analisados corretamente. Basicamente, os dados devem ser analisados de acordo com o projeto de estudo. Assim, esta a vantagem de fazer um projeto cuidadoso, bem definido do estudo é que a análise

subseqüente está normalmente clara e direta. Isto é, o projeto sugere geralmente o tipo e a extensão que a análise necessita. Entretanto, em muitos exemplos de Engenharia de Software, o projeto selecionado é complexo e o método de análise é impróprio para lidar com ele.

5. **Apresentação dos resultados**: a apresentação dos resultados é tão importante quanto a própria análise. O leitor de um estudo deve poder compreender a razão para o estudo, o projeto do estudo, a análise, os resultados e o significado dos resultados. Não somente os leitores querem aprender o que aconteceu em um estudo, mas, também querem poder reproduzir a análise ou replicar o mesmo estudo no mesmo contexto ou em contexto similar. Assim, os procedimentos de projeto e os procedimentos de levantamento de dados necessitam ser relatados em um nível de detalhe que permita que o estudo seja replicado.
6. **Interpretação dos resultados**: o objetivo principal é que todas as conclusões devem seguir diretamente dos resultados. Assim, os pesquisadores não devem introduzir material novo na seção de conclusões. É importante que os pesquisadores não deturpem suas conclusões.

No caso de banco de dados o plano de teste tem que ser realizado com mais cautela, devido à exigência de mesma *tupla*, e à persistência das variáveis.

Conforme Spoto (2000), os critérios propostos pelo autor, ao exigirem o uso da mesma *tupla* para satisfazer as associações *definição-t-uso*, distinguem-se da maioria dos demais critérios de teste; isso obriga o testador a escolher os dados de teste com mais cuidado e resulta, ao mesmo tempo, em um teste mais efetivo ao que se refere à integridade de tabelas. Estes critérios requerem o exercício de todas as possíveis combinações de usos das *queries* nos comandos SQL para cada variável tabela.

De acordo com Chays et al. (2000), quando se lida com teste de bases de dados deve-se atentar ao estado do banco de dados antes e depois da execução de um caso de teste. As alterações que ocorrem no banco de dados, a qualidade da informação armazenada, sua confiabilidade, são itens imprescindíveis para análise do banco de dados.

Em Batista (2003), tem-se que para colocar o banco de dados no estado desejado, faz-se necessário incluir, excluir ou alterar dados do banco, mantendo somente dados válidos de acordo com os domínios, com as relações e com as restrições. O banco de dados estará apto para entrar em funcionamento no momento em que os testes demonstrarem que ele oferece condições de operacionalização.

A avaliação do banco de dados será feita através da análise dos resultados obtidos após cada caso de teste, verificando-se definição e uso de cada variável persistente utilizada no caso de teste.

Este trabalho utiliza os critérios definidos por Spoto (2000) que exercitam diferentes associações, definição e uso persistente entre diferentes *variáveis tabela* de uma Base de Dados Relacional, faz uma sugestão de melhoria da escolha dos dados de teste para cada critério exercitado. Para a organização da geração dos elementos requeridos de cada critério que contribuem na melhoria da detecção de erros, foi adotada uma seqüência de numeração apresentada no Capítulo 4. Para a escolha dos casos de teste, como sugestão, foram adotadas as seguintes características:

a) Integridade referencial;

b) Integridade semântica;

c) Domínio de validade de atributos;

d) Dependência de atributos da mesma tabela;

e) Dependência de atributos de tabelas diferentes;

f) Atributos não nulos;

g) Atributos únicos.

Tais características contribuem na detecção de erros relacionados a elas visando assim melhorar a eficácia do dado de teste em situações em que apenas o critério pode possuir dados de teste que satisfazem a sua cobertura sem exercitar defeitos relacionados a essas características. Pretende-se desta forma, aumentar a eficácia dos critérios definidos por Spoto (2000) melhorando as escolhas das *tuplas* na composição dos dados de teste.

A criatividade do testador pode ser o principal instrumento de melhoria da eficácia em caso de teste, pelo fato da escolha de dados de teste, no caso específico de ABDR, onde existem vários tipos de dependências entre os atributos e entre as variáveis tabelas.

Para o experimento em questão, serão adaptadas as atividades de teste de Presman (1995), citada anteriormente e os *guidelines* de Kitchenham et al. (2002) no Capítulo 4.

3.4. Ferramenta JaBUTi

Ferramentas de auxílio ao teste de software contribuem com as execuções dos casos de testes de maneira segura e ágil. O uso de tais ferramentas de teste vem crescendo e contribuindo cada vez mais na melhoria da confiabilidade dos sistemas baseados em computador. Neste trabalho é apresentada a Ferramenta JaBUTi (*Java Bytecode Understanding and Testing*), utilizada no auxílio dos experimentos e geração dos Grafos de Programas.

Segundo Vincenzi (1998), para aplicação efetiva de um critério é necessária a existência de uma ferramenta de teste automatizada que o apóie. Através do uso de ferramentas é possível reduzir o esforço necessário para a realização do teste, bem como diminuir os erros que são causados pela intervenção humana nessa atividade. Além disso, a existência de ferramentas de teste viabiliza a realização de estudos empíricos como o objetivo de avaliar o custo e a eficácia das técnicas e critérios de teste.

Em Vincenzi et al. (2003) é apresentada uma ferramenta de teste baseada em fluxo de dados e fluxo de controle para programas e componentes Java chamada JaBUTi. A Ferramenta JaBUTi fornece ao testador diferentes critérios de teste estruturais para análise de cobertura, um conjunto de métricas estáticas para se avaliar a complexidade das classes que compõem o programa/componente, e implementa ainda algumas heurísticas de particionamento de programas que visam a auxiliar a localização de defeitos. Atualmente, essa ferramenta exercita apenas o teste de programas tradicionais, não abrangendo os critérios de teste estrutural para ABDR. Porém, essa abordagem vem sendo desenvolvida em Nardi et.al. (2005).

As principais atividades executadas pela JaBUTi, citadas por Vincenzi (2004), para realizar a análise de cobertura são: instrumentar arquivos *.class*, coletar informação de cobertura durante a execução do programa (*execute trace information*), e determinar quão bem cada um dos métodos de todas as classes foram testados de acordo com os critérios de teste disponíveis.

Vincenzi (2004), utilizou o conceito de Agrawal (1994) de dominadores e super-bloco, para facilitar a geração de casos de teste de modo a aumentar a cobertura em relação aos critérios de teste; a ferramenta atribui diferentes pesos aos requisitos de teste indicando qual o requisito de teste que, se coberto, aumentaria a cobertura em relação ao critério, considerado o máximo possível. Para avaliar o andamento da atividade de teste, relatórios de teste com diferentes níveis de granularidade (por projeto, por classe, por método, por caso de teste) podem ser gerados a fim de auxiliar o testador a decidir quando parar os testes ou quais partes ainda não foram suficientemente testadas.

Devido à portabilidade da Linguagem Java, a Ferramenta JaBUTi foi implementada completamente nesta linguagem, viabilizando a utilização da ferramenta por desenvolvedores de diferentes plataformas.

A principal característica desta ferramenta é de não requerer o código fonte para realizar o teste, podendo basear-se no código de *bytes* do programa Java, visando proporcionar a execução do teste estrutural sobre componentes e programas Java. JaBUTi permite explorar e aprender conceitos referentes ao teste de fluxo de dados e fluxo de controle. Além disso, fornece uma boa visualização da cobertura do teste aplicado.

Sendo esta uma ferramenta de teste baseada em fluxo de dados e fluxo de controle para programas e componentes Java, pretende-se apoiar todas as classes do sistema do estudo de caso deste trabalho que utilizam banco de dados relacional, criando os grafos e acompanhando os caminhos percorridos durante o teste.

4. ESTUDO DE CASO – ETAPA EXPERIMENTAL

Vários critérios de Teste de Software são propostos e elaborados, porém, poucos deles são avaliados com os tipos de erros que podem ser encontrados, principalmente em se tratando de Banco de Dados Relacional. Seguindo os guidelines de Kitchenham et al.(2002), serão apresentados: contexto e projeto experimental, administração do experimento e levantamento de dados, análise e apresentação dos resultados.

4.1. Contexto e Projeto experimental

Utilizando um sistema de aplicação em Linguagem Java com acesso ao banco de dados *Oracle9i* realizado para uma suposta empresa de cosméticos foi realizada a etapa experimental deste projeto visando a exercitar as sugestões citadas no capítulo anterior, bem como os critérios de teste estrutural para ABDR propostos por Spoto (2000) e posteriormente avaliar os resultados relacionados com a eficácia dos critérios e melhoria dos resultados com as sugestões dos dados de teste para os devidos critérios. A detecção de defeitos é possível de ser avaliada a partir dos resultados obtidos em cada caso de teste e devidamente comparados com os resultados esperados observando as especificações existentes na aplicação e no projeto do banco de dados.

Tendo em vista que os critérios de teste de ABDR definidos em Spoto (2000) são fortemente baseados na dependência de dados ocasionados por um ou mais métodos dentro de uma mesma classe ou em classes diferentes, procurou-se utilizar apenas os critérios de integração intra-classe e inter-classe nesta etapa experimental visando observar tipos de defeitos em tais dependências entre os métodos bem como entre as tabelas da base de dados.

Após os testes realizados no sistema de cosméticos, utilizando os critérios de teste de ABDR, foram realizados testes na Ferramenta JaBUTi, e seus critérios, selecionando para teste as classes que utilizam os comandos DML (Banco.java e Operacoes.java), e outras classes que se destacam do sistema como as classes CadastraClienteGUI.Java (realiza o

cadastro de clientes), CadastraVendaGUI.java (realiza as vendas de clientes) e QueryBD.java (cria as *queries* de todos os comandos DML deste sistema). Para a realização dos testes na Ferramenta JaBUTi, foram escolhidos os mesmos casos de testes gerados para o critério ABDR. Esta etapa de teste, de uma forma geral, foi realizada para comparar a cobertura que os dados de testes gerados para os critérios de Spoto, com os critérios implementados na JaBUTi (*todos-os-nós*, *todos-os-arcos*, *todos-os-usos* e *todos-os-potenciais-usos*). Essa é uma maneira de verificar se os critérios de ABDR podem ou não substituir os critérios implementados na JaBUTi para as classes que possuem comandos de SQL.

A eficácia do teste é comparada como a força de um critério em detectar defeitos de um software durante a sua execução com baixo custo (visa em auxiliar na geração de dados de teste com objetivo de detectar defeitos ainda não revelados por outros critérios).

A realização do experimento foi inicialmente planejada baseada nos códigos fontes da aplicação, bem como, nos Diagramas de Entidade e Relacionamento (DER) gerado com base nos scripts de criação de tabelas pela Ferramenta ERWIN®. Com base no código fonte da aplicação foi realizado um estudo para levantar os métodos que ocasionam definições persistentes e os métodos que ocasionam os usos das variáveis tabela (denominados de *t-uso*). Em seguida foi estabelecido um planejamento para o teste conforme Presman (1998), apresentado a seguir.

4.2. Administração do experimento e levantamento de dados

4.2.1. Preparação do Código Fonte para o início do Teste:

a) Etapas do teste:

i) Estudar o código do sistema em experimento, verificar quais classes possuem métodos que manipulam as informações no banco de dados. Para realizar esta tarefa foram utilizados para documentação inicial os Diagramas de Projeto como Diagrama de Fluxo de Dados (DFD), Use-Case, etc, bem como DER;

ii) Instrumentação do programa fonte para cada método das classes que possuem comandos **executeUpdate** ou **executeQuery** (responsáveis em definir e usar *variáveis tabelas*, respectivamentes); e geração dos grafos de programa com a Ferramenta JaBUTi (em desenvolvimento por Nardi et.al. (2005)) com as adaptações da representação gráfica do programa definido por Spoto (2000), onde os comandos DML são visualizados por um retângulo (Capítulo 3);

iii) Para a instrumentação de programas de ABDR em Java foi necessário realizar uma etapa de instrumentação do programa fonte (.java) para auxiliar na etapa de avaliação dos resultados. Para isso a instrumentação das informações necessárias foi realizada manualmente, e foram abordados os seguintes itens: todos os nós dos métodos que contêm pelo menos um comando de acesso ao SGBD (**executeUpdate** ou **executeQuey**); das informações dos comandos utilizados (*INSERT, UPDATE, DELETE ou SELECT*); das informações de qual ou quais tabelas eram envolvidas nas operações e também a tupla envolvida na operação (chaves primárias). Como exemplo na Figura 4.1, observa-se ao lado esquerdo o código do método *alteraDados()*, instrumentado conforme o grafo extraído da Ferramenta JaBUTi, ao lado direito. Destacando-se em vermelho no código as linhas com um método TesteDBpontaProvaSTR(), onde os parâmetros indicam qual método, nó, tabelas e chaves primárias que o programa está executando. Após a instrumentação foram realizados alguns ensaios para observar se os resultados (arquivos gerados na execução) atendiam ou não as expectativas.

4.2.2. Plano de Teste

i) Requisitos a serem testados ou verificados:

- Gerar os elementos requeridos pelos critérios de teste estrutural de integração intra-classe e inter-classe para todas as variáveis tabelas;

ii) Ferramentas e técnicas a serem utilizadas:

- Atualmente não existem ferramentas para utilizar teste de fluxo de dados em variáveis persistentes de ABDR, para os critérios de Spoto (2000) de maneira automatizada. A Ferramenta JaBUTi, na versão atual vem sendo modificada para atender aos critérios de teste estrutural para ABDR e já identifica comandos de SQL para o teste de unidade (NARDI et.al., 2005). Neste caso escolheu-se a Ferramenta JaBUTi por gerar grafo que identifica comandos de SQL.

```
protected boolean alterarDados(Banco bd, String query) throws SQLException {
        TesteBD.pontaProvaSTR("", "13105 00 ");
        try {
                TesteBD.pontaProvaSTR("", "07 ");
                bd.conectaBanco();
            if (bd.estadoBanco == 1) {
                Statement statement = bd.getConnection().createStatement();
                TesteBD.pontaProvaSTR("", "26 62 "+VarGlobais.gkey);
                statement.executeUpdate(query);
                TesteBD.pontaProvaSTR("", "69 ");
                connection.commit();
                TesteBD.pontaProvaSTR("", "78 86 ");
                statement.close();
             }
        else {
                TesteBD.pontaProvaSTR("", "102 110 ");
                return false;
                }
        TesteBD.pontaProvaSTR("", "111 ");
        }
        catch (SQLException sqle) {
            TesteBD.pontaProvaSTR("", "121 ");
             return false;
        }
        TesteBD.pontaProvaSTR("", "131 ");
        bd.desconectaBanco();
        return true;
}
```

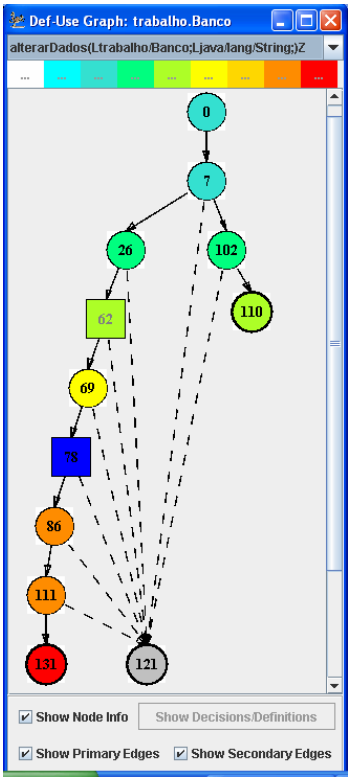


Figura 4.1 Exemplo de Instrumentação do método *alteraDados()* (13105), juntamente com o respectivo grafo gerado através da Ferramenta JaBUTi

4.2.3. Projeto de Caso de Teste:

a) Selecionar os dados de teste adequados para exercitar cada elemento requerido gerado no item anterior, observando as *tuplas* envolvidas para cada par de execuções ou observando o histórico que marca o último estado para uma determinada tabela (vide Figura 4.2). Na Figura 4.2 é apresentado o histórico do caminho percorrido no caso de teste (gerado pelo método *TesteBDpontaProvaSTR()* – linha da Figura 4.1) , sendo destacado em vermelho

a definição da *tupla* (**INS CLI CPF_CLI=111**) inserindo um cliente 111, na classe 13000 do método 104 (13104), percorrendo o caminho 00, 07, 26, **62**, 69, **78**, 86, 111, 142, onde há um comando de definição persistente nos nós 62 e 78 a persistência da definição ocorre no nó 78 comando *commit*.

b) Selecionar dados de testes possíveis de atender situações em que a Aplicação pode se defender e inibir o acesso de pontos de definições ou usos (em situações particulares).

criando path:
#01
13108 00 07 26 62 KEY:(12128 SEL CLI CPF_CLI=111)69 98 106 190 200 212 223 234 247 259 270 12129 **13104 00 07 26 *62* KEY:(12129 INS CLI CPF_CLI=111)69 *78* 86 111 142** 13108 00 07 26 62 KEY:(12128 SEL CLI CPF_CLI=111)69 107 126 146

Figura 4.2 – Exemplo do histórico do caminho (*path*) de execução do programa

4.2.4. Desenvolvimento do Teste

Pode-se transformar os casos de teste em programas, *scripts*, que exercitarão o sistema em desenvolvimento; ferramentas próprias devem ser usadas. Os casos de testes para esse experimento foram gerados manualmente tendo em vista que as interfaces do Sistema são todas através de botões.

4.2.5. Execução do Teste e Análise dos Resultados:

a) Execução dos casos de teste baseados na estratégia de teste levantada no Item 2;

b) Avaliar os caminhos percorridos em cada caso de teste, bem como as tuplas utilizadas para checar os elementos requeridos que foram cobertos, na etapa de teste em execução;

c) Em seguida, avaliar os resultados obtidos (saída do programa) e avaliação do estado das tabelas envolvidas com o objetivo de verificar se houve ou não a ocorrência de

algum defeito, com base na execução da aplicação ou nas dependências das variáveis tabelas da base de dados;

4.3. O Sistema para Teste

Para o experimento foi utilizado um sistema de controle de cosméticos, desenvolvido com as etapas previstas pela Engenharia de Software, em laboratório de uma Universidade. Existem os programas fontes, os *scripts* para criação das tabelas e os diagramas de projeto com todas as especificações. Além de ter sido utilizado etapas de Verificação e Validação (V&V) nas fases de desenvolvimento do sistema. Observa-se neste sistema, que devido ao desenvolvimento, existem muitas consistências inseridas na aplicação, sendo que estas poderiam estar no banco de dados, de forma que muitos elementos requeridos não foram cobertos devido a estas consistências, conseqüentemente um ponto positivo para o software.

No sistema em questão, consta basicamente cadastro de Clientes, Funcionário, Fornecedores, Produtos, Vendas e Compras. O mesmo foi desenvolvido em Linguagem Java com banco de dados *Oracle9i*.

Utilizando a Ferramenta JaBUTi para o teste, foram criados os grafos dos métodos que utilizam comandos DML, na qual pertencem às classes Banco.Java e Operacoes. Tais métodos foram instrumentados no programa fonte com uma função criada para instrumentação chamada TesteBD.pontaProvaSTR(<parâmetro>) na qual o parâmetro identifica qual a classe, o método e o nó que está passando (exemplo na Figura 4.2: **13104 00 07 26 *62* KEY:(12129 INS CLI CPF_CLI=111)69 *78* 86 111 142**). Se o nó for de algum comando DML, é identificado o comando DML (**INS**) que está sendo utilizado, a classe e o método em que foi gerada a SQL (**12129**), a tabela que está sendo definida ou usada (**CLI**) e, se for o caso, é identificada qual chave está sendo manipulada (**CPF_CLI=111**).

Pode-se ter uma visão geral das classes e principais métodos deste sistema, observando a Figura 4.3, destacando-se as classes Banco.Java e Operacoes.Java e também o grande fluxo entre as diversas classes.

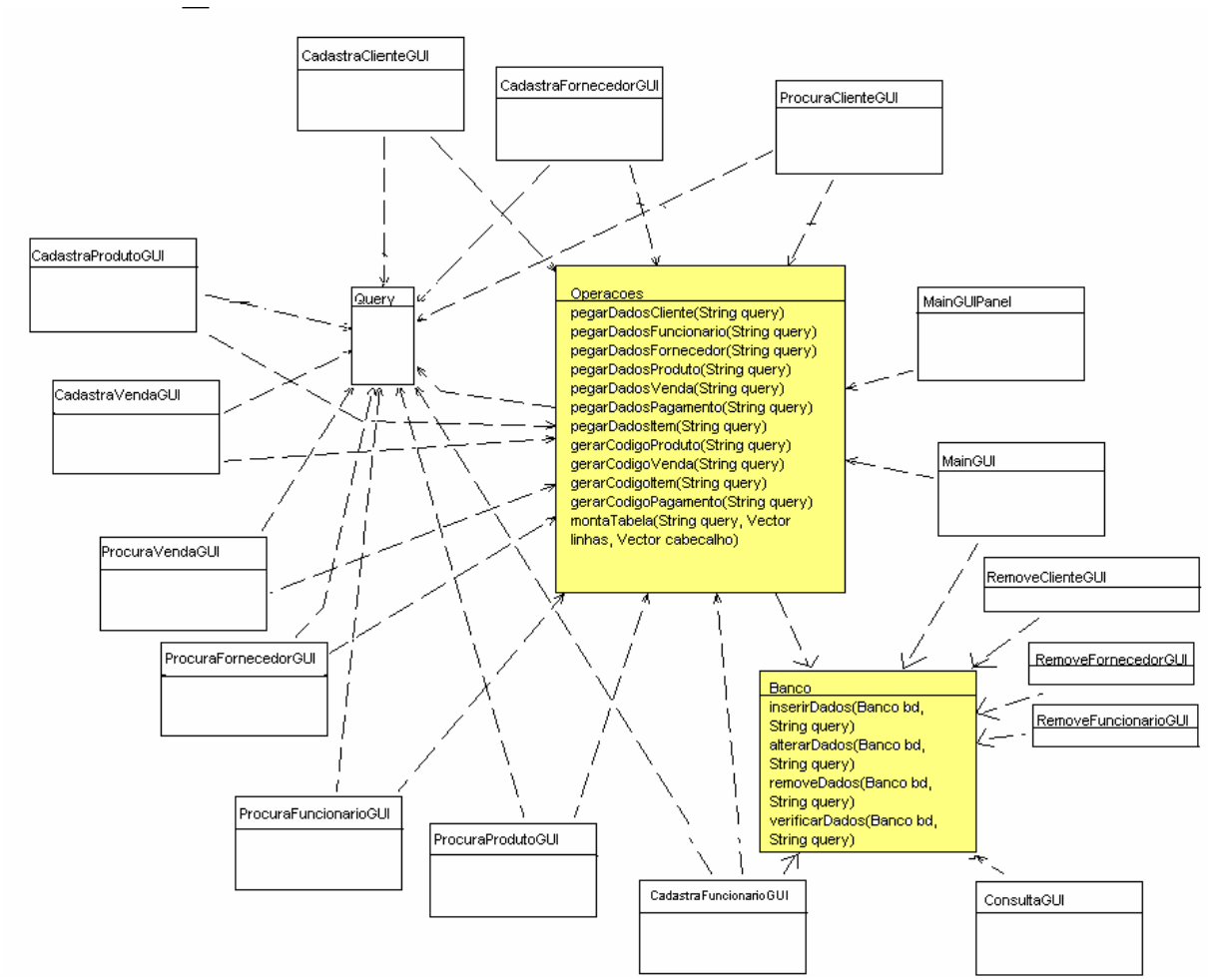


Figura 4.3 – Visão geral das classes e métodos do Sistema de Cosméticos

Para realização de uma estratégia de testes, foram numerados e organizados as os métodos e suas respectivas classes para instrumentação. A numeração da classe Banco.Java será 13000 sendo 13000 o número da classe e 100 o número do método, tendo em vista que, cada método é numerado a partir do número 100 (100 para o primeiro método, 101 para o segundo e assim sucessivamente). Por exemplo, o número atribuído para o método inserirDados é 104. Este método pertence à classe Banco.Java é de número 13000, portanto na

instrumentação do programa, quando o caminho chamar pelo método inserirDados, mostrará o número 13104 sendo a soma do número da classe com o número do método. Dentro do método inserirDados são instrumentados cada um dos nós, conforme a numeração gerada pelo grafo na Ferramenta JaBUTi (NARDI et.al., 2005). Segue um exemplo das principais classes e respectivos métodos que seguem os testes, conforme a Figura 4.4:

- **Classe: 13000 – Banco. Java**
- **Métodos**:
  - **104** - inserirDados(Banco bd, String query)
  - **105** - alterarDados(Banco bd, String query)
  - **106** - removerDados(Banco bd, String query)
  - **108** - verificarDados(Banco bd, String query)
- **Classe: 11000 - Operacoes.Java**
- **Métodos**:
  - **105** - pegarDadosCliente(String query)
  - **106** - pegarDadosFuncionario(String query)
  - **107** - pegarDadosFornecedor(String query)
  - **108** - pegarDadosProduto(String query)
  - **109** - pegarDadosCompra(String query)
  - **110** - pegarDadosVenda(String query)
  - **111** - pegarDadosPagamento(String query)
  - **112** - pegarDadosItem(String query, int n)
  - **114** - gerarCodigoProduto(String query)
  - **115** - gerarCodigoCompra(String query)
  - **116** - gerarCodigoVenda(String query)
  - **117** - gerarCodigoItem(String query)
  - **118** - gerarCodigoPagamento(String query)
  - **119** - montarTabela(String query, Vector linhas, Vector cabeçalho)

Figura 4.4 – Principais Classes e Métodos do Sistema

Na Figura 4.5 é apresentado o Diagrama de Entidade Relacionamento do sistema em questão, onde existem as tabelas: tabela_cliente, tabela_funcionário, tabela_fornecedor, tabela_produto, tabela_venda, tabela_pagamento, tabela_itens e tabela_compra. Para continuação do experimento, os nomes das tabelas serão resumidos para facilitar os testes, ficando os nomes da seguinte forma: tabela_cliente=Cli, tabela_funcionário=Func, tabela_fornecedor=Forn, tabela_produto=Prod, tabela_venda=Vend, tabela_pagamento=Pag, tabela_itens=Item e tabela_compra (não fará parte do teste).

Os grafos gerados pela Ferramenta JaBUTi, dos métodos da classes Banco.Java e Operacoes.Java, que utilizam diretamente banco de dados podem ser observados no Anexo II.

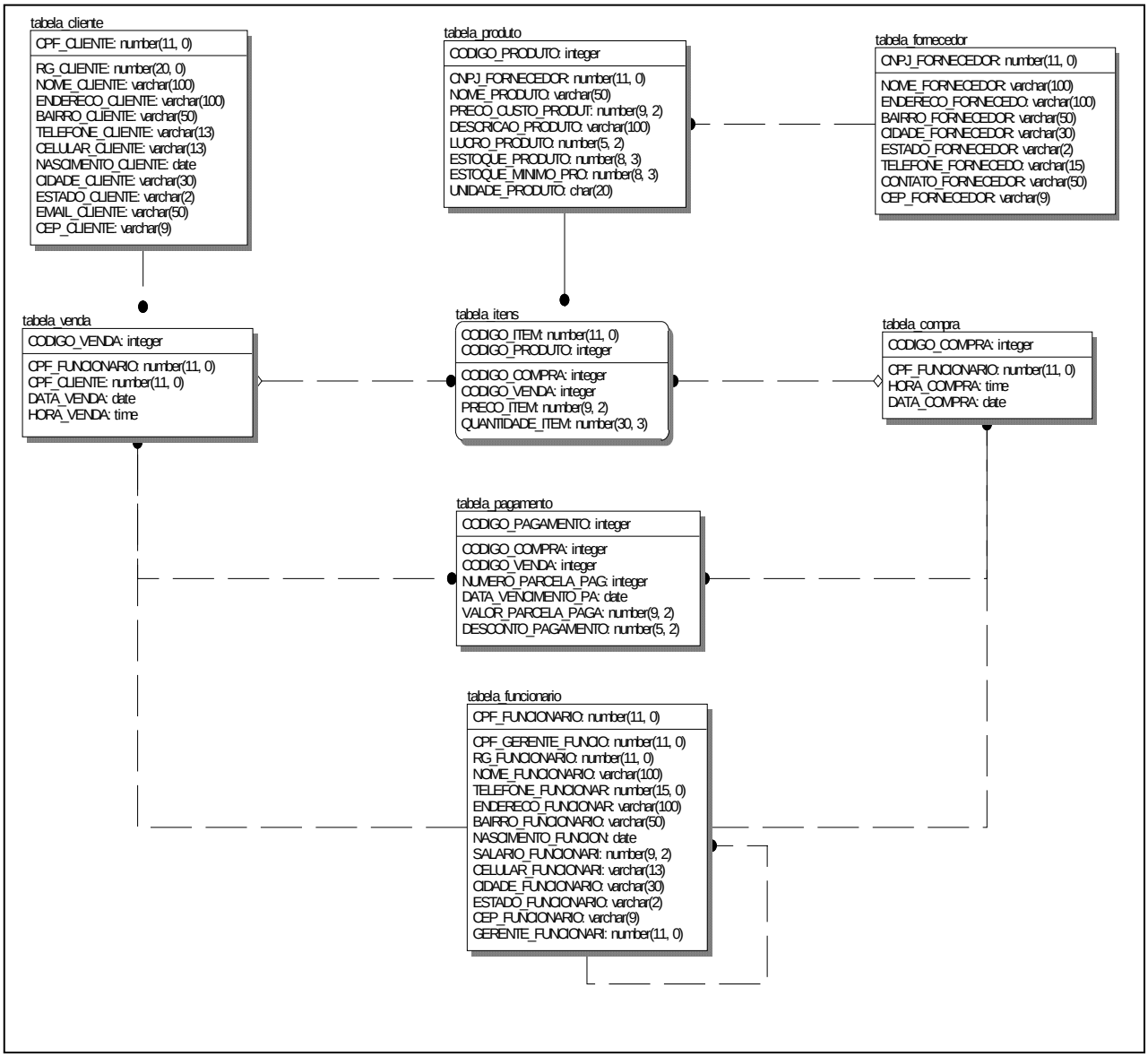


Figura 4.5: Diagrama de Entidade Relacionamento do Sistema de Cosméticos

Os elementos exigidos por um critério de teste, são os elementos requeridos (ER) e estão no Anexo I.

Para identificação dos elementos requeridos tem-se uma adaptação da definição de Spoto (2000), conforme segue:

<t, (clasmet1, <def, commit>, clasmet2, (nódeuso, nódesaída)>

sendo:

1. ***t***: a variável tabela;
2. ***clasmet1***: classe e método de definição da variável tabela;
3. ***def***: nó de definição de t (*INSERT* ou *UPDATE* ou *DELETE*);
4. ***commit***: nó que ocorre a persistência da definição (*COMMIT*);
5. ***clasmet2***: classe e método que ocorre o t-uso;
6. ***(nódeuso, nódesaída)***: arco de t-uso, arco de saída de um comando DML;

Por exemplo, observando a Figura 4.6, teremos:

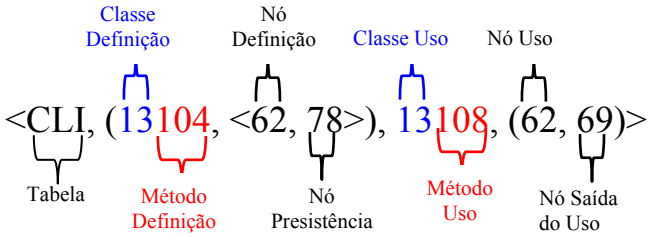


Figura 4.6 – Identificação dos Elementos Requeridos

- CLI: variável tabela;
- 13104: indica que está na classe 13000, no método 104 a definição da variável tabela CLI;
- 62: nó de definição da tabela CLI;
- 78: nó que ocorre a persistência da definição da tabela CLI (comando *commit*);
- 13108: indica a classe 13000, no método 108, ocorre um uso da variável tabela CLI;
- 62: nó de uso da variável tabela CLI;
- 69: nó de saída do uso da variável tabela CLI.

Em nenhum momento tentaremos exaurir todos os testes possíveis de uma tabela, coluna ou relacionamento. A proposta é exercitar diferentes associações definição e uso persistente entre diferentes variáveis tabela de uma Base de Dados Relacional, mostrar que um dado critério de teste exercita um conjunto de possíveis ocorrências de defeitos associado a um conjunto de características de restrições de Banco de Dados Relacional, abrangendo outras sugestões para a escolha do dado de teste, citados no Capítulo 3, que podem contribuir com outros tipos de detecção de defeitos como integridade referencial, domínio de validade de atributos, dependência de atributos da mesma tabela, dependência de atributos de tabelas diferentes, atributos não nulos, atributos únicos, tratar erros de exceção, entre outros.

4.4. Execução dos Casos de Testes

Segue a etapa de popular as tabelas através da aplicação (Sistema de Cosméticos), considerando que o banco de dados esteja vazio.

Os testes serão divididos em duas etapas, sendo que na primeira será utilizado o fluxo de dados ***intra-classe*** onde o foco foi a integração entre os métodos de uma mesma classe e a respectiva cobertura dos elementos requeridos, a classe em questão é a Banco.Java. Para a segunda etapa, foi utilizado o fluxo de dados inter-classe, sendo observada a integração entre os métodos de classes diferentes e os respectivos elementos requeridos, onde as classes são Operacoes.Java e Banco.Java. Foi observado também que o teste intra-classe para a classe Operacoes.Java torna-se insuficiente, tendo em vista que esta classe não possui pontos de definição persistente (considerados todos métodos pertencentes a esta classe). Sendo assim, apenas os critérios inter-classes é que proporcionam a geração de elementos requeridos para exercitar os métodos da classe Operaçoes.Java quando esta classe for integrada com a classe Banco.Java (onde se encontram os principais pontos de definição persistente).

A etapa de teste, inicia-se com a escolha das entradas de testes com base nos elementos requeridos dos critérios (intra-classe ou inter-classe), visando assim atender a

cobertura dos elementos requeridos dos critérios em teste. Em seguida, analisa-se a cobertura e os dados de saída.

A estratégia de definir e usar uma tabela com a mesma *tupla* torna o critério de teste mais exigente, visando uma maior atenção e organização das execuções dos casos de testes. Com as informações obtidas pela instrumentação do programa e resultados gerados durante a execução (arquivos de *paths*, tabelas em uso e *tuplas*) é realizada uma análise de cobertura dos elementos requeridos entre os casos de testes realizados. Essa análise verifica qual caminho foi percorrido na definição e uso de uma variável tabela bem como a respectiva tupla utilizada nestas execuções (definição e uso). O mecanismo de controle do caso de teste e os respectivos resultados, ao exercitar uma definição e um uso com relação a uma dada tabela, foi utilizar uma instrumentação do número de quais classes e métodos que foram envolvidos durante a execução dos casos de teste. Seguem alguns exemplos de testes realizados:

Exemplo 1- Intra-Classe:

Para mostrar a cobertura do elemento requerido seis, (será padronizado para **ER6**, para referir-se ao elemento requerido e seu respectivo número) abaixo, foi necessário realizar dois casos de testes sendo o primeiro que executa o comando Insert e o segundo que executa o comando Delete, para o ER6:

- **ER6 -** <CLI, (13104, <62, 78>), 13106, (62, 136)>

Definição: Inserindo cliente

Uso: Excluir o mesmo cliente, percorrendo o caminho de exceção (*catch*).

Dados de teste (entradas do programa):

- **Caso de Teste 12**: Inserir Cliente 1;
- **Caso de Teste 13**: Inserir uma venda com Cliente 1;
- **Caso de Teste 14**: Excluir o cliente 1;

- **Qual o comportamento esperado:** Como existe uma dependência de vendas com cliente, espera-se que com este teste, no momento de excluir o cliente com esta dependência, percorra o caminho de exceção;

- Caminho percorrido no arquivo texto gerado ER006ct12_13_14.TXT (obs: padronizado o arquivo gerado da seguinte forma: ER006: Elemento Requerido 6, ct12_13_14: caso de teste 12, 13 e 14).

- **Definindo**:

 # 12: 13104 00 07 26 **62** 69 **78** 86 111 142

Chave: KEY:(12129 INS CLI CPF_CLI=1) (obs: o Número 12129, significa que esta *query* foi criada na classe 12000, método 129).

- **Usando**:

 # 14: 13106 00 07 26 **62** 69 92 **100** 108 160

Chave: KEY:(12124 DEL CLI CPF_CLI=1) (obs: o Número 12124, significa que esta *query* foi criada na classe 12000, método 124).

- **Qual cobertura obtida dos elementos requeridos**: Não foi possível cobrir este elemento requerido, devido às consistências do próprio sistema de cosméticos. Foi definido o cliente (inserindo), logo após foi realizada uma venda deste cliente. No uso do cliente para o elemento requerido em questão (excluir o cliente), o sistema mostra uma mensagem:"Há vendas deste cliente. Remova-as e tente novamente." Logo após, a aplicação mostra a venda deste cliente com a opção de remover. Foi removida a venda. A aplicação volta para a opção de remover o cliente. É selecionado o cliente 1 e botão remover. O sistema remove o cliente sem percorrer o caminho de *catch*.

Este elemento requerido foi separado e analisado posteriormente. Após a análise, foi criada uma tabela **Promocao** com uma dependência de cliente apenas no banco de dados para efeito de teste. Logo após foi realizado um novo teste para cobrir o elemento requerido **ER6**,

mas como existirá agora uma restrição de integridade referencial entre cliente e promoção ativa ele não permitirá a exclusão do cliente, executando assim o caminho de erro (*catch*). Isto mostra que o critério é efetivo na detecção desse tipo de defeito.

- **Caso de Teste 592:** Inserir Cliente 2;

- Inserir via *SQL*Plus* do *Oracle 9i*, uma tupla na tabela **Promocao** com referência ao código do Cliente 2; (*create table Promoção (codigo number(11,0) not null, cpf_cliente integer); alter table Promocao add foreign key (cpf_cliente) references tabela_cliente (cpf_cliente)*);

- **Caso de Teste 593:** Excluir o cliente 2;

- **Qual o comportamento esperado**: Como existe uma integridade referencial (dependência) entre **Promocao** e cliente, espera-se que com este teste, no momento de excluir o cliente com esta dependência, percorra o caminho de exceção (já que a aplicação desconhece tal dependência), cobrindo o **ER6**. E, com isso, mostrar que sempre que a aplicação ou o banco de dados deixar de avaliar uma dependência entre tabelas o critério será eficaz em detectar tais erros.

- **Caminho percorrido** no arquivo gerado ER006ct_B_592_593.TXT.

- **Definindo**:

592: 13104 00 07 26 **62** 69 **78** 86 111 142

Chave: KEY:(12129 INS CLI CPF_CLI=2)

- **Usando**:

 # 593: 13106 00 07 26 **62 136**

 # Chave - KEY:(12124 DEL CLI CPF_CLI=2)

- **Qual comportamento obtido**: O sistema emitiu uma mensagem de erro: Erro ao remover o cliente. Botão “ok”. Analisando o caminho percorrido, pode-se afirmar que o

elemento requerido foi coberto e com isso o critério foi eficaz em detectar erros de dependência, caso a aplicação não possuir defesas.

Exemplo 2- Intra-Classe Ciclo 2:

- Cobertura de ER143 esperada:

- **ER143.** <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (13106, (62, 100))>

Definição: Inserir Fornecedor 888

Definição: Inserir Produto 16 com dependência do fornecedor 1

Uso: Excluir o produto 16

- Dados de teste (entradas do programa):

- **Caso de Teste 400**: Inserir Fornecedor 888;

- **Caso de Teste 401**: Inserir um Produto (16) com dependência do Fornecedor 888;

- **Caso de Teste 402**: Excluir o Produto 16;

- **Qual o comportamento esperado**: Como existe uma dependência de produto com fornecedor, espera-se que inserindo um fornecedor possa inserir um produto com referência ao fornecedor que acabou de ser cadastrado, ou seja, permitir inserir o fornecedor 888, inserir o produto 16 com dependência do fornecedor 888 e excluir o produto 16;

- Utilizando as sugestões deste trabalho, será verificado o dado do produto como preço de custo espera-se que seja maior que zero;

- **Caminho percorrido** no arquivo gerado ER143ct_400_401_402.TXT.

- **Definindo**:

 # 400: 13104 00 07 26 **62** 69 **78** 86 111 142;

Chave: KEY:(12135 INS FORN CNPJ_FORN=888);

- **Definindo**:

 # 401 - 13104 00 07 26 **62** 69 **78** 86 111 142;

Chave: KEY:(12131 INS PROD COD_PROD=16);

- **Usando**:

 # 402 - 13106 00 07 26 **62** 69 92 **100** 108 160;

Chave: KEY:(12127 DEL PROD COD_PRO=16);

- **Qual a cobertura de elemento requerido é obtida**: O elemento requerido foi coberto com sucesso.

- **Qual o comportamento obtido**: Apesar do elemento percorrido ser exercitado com sucesso, a aplicação permitiu inserir um produto com valor de custo menor que zero.

- Foi aplicado o critério intra-classe-ciclo2, pois, existe a necessidade de se definir uma variável tabela (fornecedor) para conseguir a associação definição-t-uso de outra variável tabela (produto). Este critério é eficaz em detectar erros de dependência, além de detectar se a segurança das transações entre os comandos de manipulação da base de dados está correta neste exemplo, o comando delete (de produto) tem uma regra relacionada à tabela de itens – caso exista um item deste produto, a aplicação não deve permitir a exclusão da *tupla*;

- Com as sugestões deste trabalho foi detectado problema de validade de atributos, no caso o atributo preço de custo da variável tabela produto.

Exemplo 3- Intra-Classe Ciclo 2:

- Cobertura de elemento requerido esperada para o **ER179**:

- **ER179.** <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), ITEM, (13106, (62, 100))>;

Definição: Inserir Venda 46

Definição: Inserir Item 39 com dependência de Venda e Produto

Uso: Excluir Item 39

Obs: o código de venda e o código de item são automáticos.

- **Dados de teste (entradas do programa):**

- **Caso de Teste 514**: Inserir Venda 46;

- **Caso de Teste 515**: Inserir Item 39 com dependência de Venda e Produto;

- **Caso de Teste 516**: Excluir Item 39;

- **Qual o comportamento esperado**: Como existe uma dependência da tabela item com as tabelas venda e produto, espera-se que inserindo uma venda possam ser inseridos itens com referência a venda que acabou de ser cadastrada.

- Permitir excluir o item;

- Utilizando as sugestões deste trabalho, serão verificados:

o Data de venda: tem que ser data atual (sugestão: domínio de validade de atributos);

o Preço do item: maior que zero (sugestão: domínio de validade de atributos);

o Se foi efetuada a baixa do estoque, na variável tabela produto, devido a esta venda (sugestão: dependência de atributo de outra classe);

- **Caminho percorrido** no arquivo gerado ER179ct_514_515_516.TXT;

- **Definindo**:

 # 514 - 13104 00 07 26 **62** 69 **78** 86 111 142;

Chave: KEY:(12137 INS VEND COD_VEND=46);

- **Definindo**:

 # 515 - 13104 00 07 26 **62**) 69 **78** 86 111 142;

Chave: KEY:(12142 INS ITEM COD_ITEM=39 COD_PROD=10;

- **Usando**:

 # 516: 13106 00 07 26 **62** 69 92 **100** 108 160;

 # Chave: KEY:(12102 DEL ITEM COD_VEND 46);

- **Com relação à cobertura do elemento requerido ER179**: O elemento requerido foi coberto com sucesso.

- **Qual o comportamento obtido**: Apesar do elemento percorrido ser coberto, a aplicação permitiu: a) inserir uma venda com data inválida; b) inserir um item com valor menor que zero; c) não efetuou baixa da quantidade do estoque;

- O critério intra-classe-ciclo2, foi aplicado, pois, existe a necessidade de se definir uma variável tabela (venda) para conseguir a associação definição-t-uso de outra variável tabela (itens). Este critério é eficaz em detectar erros de dependência, pois, há a necessidade de ter uma venda, para inserir o item.

Com as sugestões deste trabalho, para este teste, foram detectados problemas de domínio de validade de atributo e dependência de atributos de variáveis tabela diferentes (quando efetuada uma venda de um produto, efetuar baixa em estoque), atributo quantidade_item da tabela itens, com o atributo estoque_produto da variável tabela produto.

Com essa pequena amostra de execuções de casos de teste para atender os elementos Requeridos do critério *todos-t-usos-Ciclo1* e *todos-t-usos-Ciclo2* (ambos *intra-classe*) pode-se observar que a exigência da atenção e a necessidade de um bom plano de teste são fundamentais para o bom desempenho do teste em busca da detecção de erros baseados nas dependências de dados nas tabelas e na Aplicação quando não houver detalhes de verificação das dependência de dados existentes nas tabelas.

4.5. Análise e Apresentação dos Resultados

A avaliação de cobertura de elementos requeridos pelos critérios *intra-classe* e *inter-classe* foi realizada, com ajuda de planilha eletrônica Microsoft Excel. Os elementos requeridos não exercitados foram separados e examinados posteriormente existindo a possibilidade de inserir novos casos de teste, ou incluir alguma dependência para exercitar tais elementos requeridos não cobertos (para os casos com erros). Como por exemplo, o elemento requerido ER6. <CLI, (13104, <62, 78>), 13106, (62, 136)>, mostrado na seção 4.3, exemplo 1, em que existe uma definição persistente ao inserir um cliente 13104, (<62, 78>) e um *t-uso*

da tabela cliente ao excluir a tabela cliente 13106, (62, 136), mas, com a saída do arco para o nó que possui o comando de tratamento de exceção (*catch*). Neste caso foi necessário incluir uma tabela dual (tabela para teste), que possui uma dependência com a tabela cliente, sem o conhecimento devido da aplicação, forçando assim a ocorrência de uma falha na exclusão da tabela cliente, obrigando o fluxo após a exclusão passar pelo comando de tratamento de erro (*catch*). Essa inclusão de uma tabela com dependência só foi necessária após todas tentativas de execução sem sucesso na cobertura deste elemento requerido. Com a cobertura do deste elemento requerido verificou-se que o critério é eficaz em detectar erros de dependência quando a aplicação não possuir a defesa de tais dependências.

Observou-se que existem elementos requeridos que não são exercitados com a mesma tupla; esses elementos podem ser descartados após algumas tentativas (SPOTO, 2000). Há necessidade de tentar exercitar estes elementos requeridos, para verificar situações que podem revelar defeitos como exemplo o elemento requerido ER1. <CLI, (13104, <62, 78>), 13104, (62, 78)>, em que existe uma definição persistente ao inserir um cliente e uma "tentativa" de inserir o mesmo cliente, sendo que, se isso for possível com a mesma *tupla*, o critério irá detectar um erro de chave primária. Na execução deste caso de teste espera-se que para cobrir este elemento requerido, não seja possível utilizar a mesma *tupla*, sendo assim a análise de cobertura obriga o testador a observar se os fatos esperados ocorreram com sucesso ou não.

Foram detectados 404 elementos requeridos, sendo 204 intra-classe e 200 inter-classes. Para a cobertura de casos de teste intra-classe, foram realizados 627 casos de testes, sendo que foram necessários 497 casos de testes para cobrir 154 elementos requeridos do *critério todos-t-uso-ciclo1-intra* de um total de 172 elementos requeridos, resultando em uma cobertura de 89, 53%. Para o *critério todos-t-uso-intra-ciclo2*, na qual existem 32 elementos

requeridos, foram realizados 130 casos de testes cobrindo 28 elementos requeridos, resultando em 87,50% de cobertura.

Para o *critério todos-t-uso-inter-ciclo1,* existem 164 elementos requeridos, foram realizados 474 casos de testes, e foram cobertos 84 elementos requeridos resultando em uma cobertura de 51,22%. Para o *critério todos-t-uso-inter-ciclo2*, na qual existem 36 elementos requeridos, foram realizados 130 casos de testes cobrindo 18 elementos requeridos, resultando em 50,00% de cobertura.

A Tabela 4.1, apresenta os resultados de cobertura do critério intra-classe *todos-t-uso-ciclo1-intra*, onde se pode observar a porcentagem de cobertura de elementos requeridos separadamente, para cada tabela, além do total exercitado pela mesma tupla, total exercitado em tuplas diferentes, total exercitado (mesma *tupla* + *tuplas* diferentes) e total não exercitado (*ne*). Assim como na Tabela 4.1, nas Tabelas 4.2 e 4.3, também são apresentados os resultados de cobertura, diferenciando que na Tabela 4.2 é apresentada a cobertura do *critério todos-t-uso-ciclo2-intra* e na Tabela 4.3 é apresentada uma visão geral da soma das Tabelas 4.1 e 4.2.

O critério de teste *todos-t-uso-ciclo1-intra*, foi eficaz em detectar presença de dependências de tabelas, além de defeitos em comandos de consultas da SQL (uso de *queries* indevidas) consistências de manipulação da SQL em relação a um atributo das variáveis tabela, por exemplo, no elemento requerido ER117. <PROD, (13104, <62, 78>), PROD, (13105, (62, 78))>, em que se encontra uma inserção na variável tabela Produto e uma alteração deste Produto. Neste caso há uma chave estrangeira para Fornecedor e se o programador permitisse alteração desta chave estrangeira e não existisse o Fornecedor correspondente, o critério acusaria um **erro de consistência**. Pode-se verificar também com este critério, se a **segurança das transações** entre os comandos de manipulação da base de dados está correta, como no caso do elemento requerido ER47. <FUNC, (13104, <62, 78>),

13106, (62, 100)>, em que existe a definição inserindo um funcionário e a exclusão deste funcionário. No caso deste sistema, o funcionário pode gerenciar outro funcionário, existindo um auto-relacionamento e o funcionário pode realizar uma venda. Para a segurança da transação, no momento de excluir o funcionário, e se o funcionário em questão estiver gerenciando outros funcionários ou ter realizado uma venda, o sistema não deveria permitir a exclusão desta *tupla*.

Tabela 4.1 - Tabela de cobertura do critério de *todos-t-uso-ciclo1-intra*

INTRA-CLASSE CICLO 1										
Tabela	**Total de Elemento Requerido Ciclo1**	**Total de Casos de Testes para CCIntra1**	**Total Exercitado Mesma Tupla**		**Total Exercitado Tuplas Diferentes (afins) - Não Executáveis**		**Total Exercitado**		**Total Não Exercitado**	
			Tot CC Intra1	**% Total CC Intra1**	**Tot CC Intra1**	**% CC Intra1**	**Tot CC Intra1**	**% CC Intra1**	**Tot CC Intra1**	**% CCIntra1**
CLI	42	118	28	66,67%	8	19,05%	36	85,71%	6	14,29%
FUNC	42	119	28	66,67%	8	19,05%	36	85,71%	6	14,29%
FORN	30	93	22	73,33%	8	26,67%	30	100,00%	0	0,00%
PROD	24	75	16	66,67%	8	33,33%	24	100,00%	0	0,00%
VEND	20	56	12	60,00%	4	20,00%	16	80,00%	4	20,00%
ITEM	6	16	3	50,00%	2	33,33%	5	83,33%	1	16,67%
PAG	8	20	5	62,50%	2	25,00%	7	87,50%	1	12,50%
TOTAL	**172**	**497**	**114**	**66,28%**	**40**	**23,26%**	**154**	**89,53%**	**18**	**10,47%**

O critério de teste *todos-t-uso-ciclo2-intra*, foi eficaz em detectar defeitos de dependências múltiplas, podendo ser aplicado quando existe a necessidade de se definir uma tabela para conseguir a *associação-definição-t-uso* de outra variável tabela, além de detectar erros de consistências e segurança de transações, assim como no critério *todos-t-uso-ciclo1-intra*.

Tabela 4.2- Tabela de cobertura do critério de *todos-t-uso-ciclo2-intra*

INTRA-CLASSE CICLO 2										
Tabela	**Total de Elemento Requerido Ciclo2**	**Total de Casos de Testes para CCIntra2**	**Total Exercitado Mesma Tupla**		**Total Exercitado Tuplas Diferentes (afins) - Não Executáveis**		**Total Exercitado**		**Total Não Exercitado**	
			Tot CC Intra2	**% Total CC Intra2**	**Tot CC Intra2**	**% CC Intra2**	**Tot CC Intra2**	**% CC Intra2**	**Tot CC Intra2**	**% CC Intra2**
CLI	0	0	0	0,00%	0	0,00%	0	0,00%	0	0,00%
FUNC	0	0	0	0,00%	0	0,00%	0	0,00%	0	0,00%
FORN	0	0	0	0,00%	0	0,00%	0	0,00%	0	0,00%
PROD	8	29	6	75,00%	2	25,00%	8	100,00%	0	0,00%
VEND	10	47	6	60,00%	2	20,00%	8	80,00%	2	20,00%
ITEM	6	21	3	50,00%	2	33,33%	5	83,33%	1	16,67%
PAG	8	33	5	62,50%	2	25,00%	7	87,50%	1	12,50%
TOTAL	**32**	**127**	**20**	**62,50%**	**8**	**25,00%**	**28**	**87,50%**	**4**	**12,50%**

Tabela 4.3- Tabela de cobertura do Ciclo 1 X Ciclo 2 (Intra)

INTRA-CLASSE – TOTAL GERAL – Ciclo 1 X Ciclo 2										
Tabela	**Total de Elemento Requerido**	**Total de Casos de Testes**	**Total Exercitado Mesma Tupla**		**Total Exercitado Tuplas Diferentes (afins) - Não Executáveis**		**Total Exercitado**		**Total Não Exercitado (*ne*)**	
			Total Geral	**% Total Geral**	**Total Geral**	**% Total Geral**	**Total Geral**	**% Total Geral**	**Total Geral**	**% Total Geral**
CLI	42	118	28	66,67%	8	19,50%	36	85,71%	6	14,29%
FUNC	42	119	28	66,67%	8	19,05%	36	85,71%	6	14,29%
FORN	30	93	22	73,33%	8	26,67%	30	100,00%	0	0,00%
PROD	32	104	22	68,35%	10	31,25%	32	100,00%	0	0,00%
VEND	30	103	18	60,00%	6	20,00%	24	80,00%	6	20,00%
ITEM	12	37	6	50,00%	4	33,33%	10	83,33%	2	16,67%
PAG	16	53	10	62,50%	4	25,00%	14	87,50%	2	12,50%
TOTAL	**204**	**627**	**134**	**65,69%**	**48**	**25,53%**	**182**	**89,22%**	**22**	**10,78%**

Assim, como em Spoto (2000), notou-se que durante os testes, a exigência da mesma *tupla* para satisfazer os critérios de integração faz com que exercite diferentes tipos de combinações de atributos das variáveis tabela, forçando o entendimento das condições do projeto de Banco de Dados (consistência da base de dados em relação às chaves primárias e estrangeiras; restrições ou regras existentes no projeto, entre outros), sendo eficaz na detecção de defeitos de implementação dos comandos SQL.

Como já abordado anteriormente, no sistema de cosméticos existem muitas consistências inseridas na aplicação e conseqüentemente, alguns elementos requeridos foram considerados não executáveis, tendo em vista que a aplicação possui consultas automáticas não possibilitando a escolha dos dados de consulta. Foi observado principalmente nos testes inter-classe, pois, para este sistema a classe Operações.Java existem somente consultas e muitas são internas sem o acesso ao usuário.

Tabela 4.4 - Tabela de cobertura do critério de *todos-t-uso-ciclo1-inter*

INTER-CLASSE CICLO 1										
Tabela	**Total de Elemento Requerido Ciclo1**	**Total de Casos de Testes para CCInter1**	**Total Exercitado Mesma Tupla**		**Total Exercitado Tuplas Diferentes (afins) - Não Executáveis**		**Total Exercitado**		**Total Não Exercitado**	
			Tot CCInter1	**% Total CCInter1**	**Tot CCInter1**	**% CCInter1**	**Tot CCInter1**	**% CCInter1**	**Tot CCInter1**	**% CCInter1**
CLI	36	119	14	38,89%	6	16,67%	20	55,56%	16	44,44%
FUNC	42	124	13	30,95%	8	19,05%	21	50,00%	21	50,00%
FORN	24	71	8	33,33%	4	16,67%	12	50,00%	12	50,00%
PROD	24	62	8	33,33%	4	16,67%	12	50,00%	12	50,00%
VEND	20	51	5	25,00%	5	25,00%	10	50,00%	10	50,00%
ITEM	8	20	0	0,00%	4	50,00%	4	50,00%	4	50,00%
PAG	10	27	0	0,00%	5	50,00%	5	50,00%	5	50,00%
TOTAL	**164**	**474**	**48**	**29,27%**	**36**	**21,95%**	**84**	**51,22%**	**80**	**48,78%**

Na Tabela 4.4, apresenta os resultados de cobertura do "critério"*: inter-classe todos-t-uso-ciclo1-inter*, onde se pode observar a porcentagem de cobertura de elementos requeridos separadamente, para cada tabela, além do total exercitado pela mesma *tupla*, total exercitado em *tuplas* diferentes, total exercitado (mesma *tupla* + *tuplas* diferentes) e total não exercitado (*ne*). Assim como na Tabela 4.4, nas Tabelas 4.5 e 4.6 também são apresentados os resultados de cobertura, diferenciando que na Tabela 4.5 é apresentada a cobertura do *critério todos-t-uso-ciclo2-inter* e na Tabela 4.6 é apresentada uma visão geral da soma das Tabelas 4.4 e 4.6, focando os critérios inter-classe.

Os critérios de teste *todos-t-uso-ciclo1-inter* e *todos-t-uso-ciclo2-inter* foram necessários, pois, a classe Operacoes.java, não realiza definições de variáveis tabela, somente uso (consultas), necessitando assim a dependência da classe Banco.java, ou seja, esses critérios exercitam associações de variáveis tabela que são definidas na classe Banco.java são usadas na classe Operacoes.java. A eficácia destes critérios está também, assim como os critérios intra-classe, em consistências de manipulação da SQL em relação a um atributo das variáveis tabela, consistências de segurança de transações, detectar defeitos de consultas da SQL e defeitos de dependências múltiplas. A única diferença é que a definição está em uma classe e o uso está em outra classe. O percentual de cobertura dos critérios inter-classe, foi de 51,22% de um total de 200 elementos requeridos, pois, alguns pontos das consultas SQLs da classe Operações.Java, não existia acesso ao usuário na escolha do dado, pois, eram consultas de consistência e interna ao programa.

Tabela 4.5 - Tabela de cobertura do critério de *todos-t-uso-ciclo2-inter*

INTER-CLASSE CICLO 2										
Tabela	**Total de Elemento Requerido Ciclo2**	**Total de Casos de Testes para CCInter2**	**Total Exercitado Mesma Tupla**		**Total Exercitado Tuplas Diferentes (afins) - Não Executáveis**		**Total Exercitado**		**Total Não Exercitado**	
			Tot CCInter2	**%**	**Tot CCInter2**	**%**	**Tot CCInter2**	**% CCInter2**	**Tot CCInter2**	**Tot**
CLI	0	0	0	0,00%	0	0,00%	0	0,00%	0	0,00%
FUNC	0	0	0	0,00%	0	0,00%	0	0,00%	0	0,00%
FORN	0	0	0	0,00%	0	0,00%	0	0,00%	0	0,00%
PROD	8	29	4	50,00%	0	0,00%	4	50,00%	4	50,00%
VEND	10	43	5	50,00%	0	0,00%	5	50,00%	5	50,00%
ITEM	8	26	4	50,00%	0	0,00%	4	50,00%	4	50,00%
PAG	10	32	5	50,00%	0	0,00%	5	50,00%	5	50,00%
TOTAL	**36**	**130**	**18**	**50,00%**	**0**	**0,00%**	**18**	**50,00%**	**18**	**50,00%**

Tabela 4.6- Tabela de cobertura do Ciclo 1 X Ciclo 2 (Inter)

INTER-CLASSE – TOTAL GERAL – Ciclo 1 X Ciclo 2										
Tabela	Total de Elemento Requerido Inter-Classe	Total de Casos de Testes	Total Exercitado Mesma Tupla		Total Exercitado Tuplas Diferentes (afins) - Não Executáveis		Total Exercitado		Total Não Exercitado	
			Total Geral	% Total Geral	Total Geral	% Total Geral	Total Geral	% Total Geral	Total Geral	% Total Geral
CLI	36	119	14	38,89%	6	16,67%	20	55,56%	16	44,44%
FUNC	42	124	13	30,95%	8	19,05%	21	50,00%	21	50,00%
FORN	24	71	8	33,33%	4	16,67%	12	50,00%	12	50,00%
PROD	32	91	12	37,50%	4	12,50%	16	50,00%	16	50,00%
VEND	30	94	10	33,33%	5	16,67%	15	50,00%	15	50,00%
ITEM	16	46	4	25,00%	4	25,00%	8	50,00%	8	50,00%
PAG	20	59	5	25,00%	5	25,00%	10	50,00%	10	50,00%
TOTAL	**200**	**604**	**66**	**33,00%**	**36**	**18,00%**	**102**	**51,00%**	**98**	**49,00%**

Os casos de testes foram realizados com base em:

- Dados de teste (entradas do programa);
- Qual cobertura de elementos requeridos esperada;
- Qual o comportamento esperado;
- Qual cobertura de elementos requeridos obtida;
- Qual comportamento obtido.

Na elaboração destes casos de teste foram utilizadas as sugestões desta dissertação para a escolha de dados de teste refinados, e em muitos casos de teste, apesar de cobrir os elementos requeridos, nas Tabelas 4.7, 4.8, 4.9, 4.10 e 4.11 são apresentados os defeitos detectados com as sugestões de entradas de teste.

Problemas detectados que apesar de percorrer elemento requerido, não estava consistente na aplicação:

1. Integridade referencial;
2. Integridade semântica, Tamanho;
3. Tipo de dados ou Domínio de validade de atributos;
4. Dependência de atributos da mesma tabela;

5. Dependência de atributos de tabelas diferentes;

6. Atributos não nulos;

7. Atributos únicos.

As entradas de dados sugeridas, anteriormente, foram extraídas do teste funcional (validação de domínio de atributos) (PRESMAN, 1998) e adaptado para tipos de domínios e dependências em banco de dados relacionais, reforçando a eficácia de detecção de erros na aplicação com relação à entrada de dados, bem como exercitando elementos requeridos dos critérios intra e inter-classe quando estes requerem o *t-uso* em tratamento de exceção. Apenas em casos cujas defesas são rígidas na aplicação foram utilizadas a estratégia de criação de tabelas com dependências (em relação às variáveis em prática) na etapa de teste. Esta abordagem não se aplica em testes reais.

Tabela 4.7 – Defeitos detectados na Tabela Cliente

CLIENTE			
	Atributo Exercitado	**Casos de Testes com:**	**Tipo de Problema**
1	Cpf	chave primária:aceita zero, e números negativos	Tamanho, Tipo de dados ou Domínio de validade de atributos
2	Data de Nascimento	aceita data maior que data atual	Tamanho, Tipo de dados ou Domínio de validade de atributos
3	Data de Nascimento	quando 29/02/1991 - mostra msg do oracle	Tratar data de entrada
4	RG	aceita RG repetido de outro cliente	Atributos únicos
5	Estado	quando tamanho campo maior que 2 mostra msg do oracle	Tratar mensagem de exceção
6	Cep	aceita letras	Tamanho, Tipo de dados ou Domínio de validade de atributos
7	Celular	aceita letras	Tamanho, Tipo de dados ou Domínio de validade de atributos
8	Telefone	aceita letras	Tamanho, Tipo de dados ou Domínio de validade de atributos

Tabela 4.8 – Defeitos detectados na Tabela Funcionário

FUNCIONARIO			
	Atributo Exercitado	**Casos de Testes com:**	**Tipo de Problema**
1	Cpf	chave primária-aceita zero, e números negativos	Tamanho, Tipo de dados ou Domínio de validade de atributos
2	Data de Nascimento	aceita data maior que data atual	Tamanho, Tipo de dados ou Domínio de validade de atributos
3	Data de Nascimento	quando 29/02/1991 - mostra msg do oracle	Tratar mensagem de exceção
4	RG	aceita RG repetido de outro cliente	Atributos únicos
5	Estado	quando tamanho campo maior que 2 mostra msg do oracle Aceita nulo	Tamanho, Tipo de dados ou Domínio de validade de atributos Tratar mensagem de exceção
6	Cep	aceita letras/aceita nulo	Tamanho, Tipo de dados ou Domínio de validade de atributos Atributos não nulos,
7	Celular	aceita letras	Tamanho, Tipo de dados ou Domínio de validade de atributos
8	Telefone	aceita letras	Tamanho, Tipo de dados ou Domínio de validade de atributos
9	Salário	aceita valor maior que salário de gerente	Integridade semântica,
10	Salário	aceita valor negativo	Tamanho, Tipo de dados ou Domínio de validade de atributos
10	Gerente	aceita ter o mesmo como gerente	Integridade semântica,
11	Gerente	qdo mesmo como gerente há problemas na exclusão	Integridade semântica, Integridade referencial
12	Gerente	qdo a tabela estava vazia não era possível colocar algum gerente- SQL errada	Integridade semântica,
13	Exclusão de func	problemas quando existe funcionário em vendas e for gerente	Integridade semântica,
14	Cidade	aceita nulo	Atributos não nulos,

Tabela 4.9 – Defeitos detectados na Tabela Fornecedor

FORNECEDOR			
	Atributo Exercitado	**Casos de Testes com:**	**Tipo de Problema**
1	CNPJ	chave primária - aceita zero, e números negativos, tamanho (11) de cnpj menor que o normal	Tamanho, Tipo de dados ou Domínio de validade de atributos
2	Estado	quando tamanho campo maior que 2 mostra msg do oracle	Tamanho, Tipo de dados ou Domínio de validade de atributos Tratar mensagem de exceção
3	Cep	aceita letras	Tamanho, Tipo de dados ou Domínio de validade de atributos

Tabela 4.10 – Defeitos detectados nas Tabelas Venda, Itens e Pagamento

VENDA/ITEM/PAGAMENTO			
	Atributo Exercitado	**Casos de Testes com:**	**Tipo de Problema**
1	Codigo Venda	código automático - ok	
2	Codigo Item	código automático-?	Sugestão de chave primária Código_venda + Código_Produto
3	Codigo Pagamento	código automático-?	Sugestão de chave primária Código_venda + Num_Parcela
4	Cliente	quando cadastra novo cliente em venda, não aparece no grid	Sugestão atualizar grid
5	Data Venda	Aceita data menor que atual e maior que atual(obs: principalmente ano)	Tamanho, Tipo de dados ou Domínio de validade de atributos
6	Preço Item	Aceita preço negativo	Tamanho, Tipo de dados ou Domínio de validade de atributos
7	Desconto Pagamento	quando digita desconto, não altera o valor da parcela/somente se insere um novo item	Tamanho, Tipo de dados ou Domínio de validade de atributos Dependência de atributos da mesma tabela
8	Parcelas	quando altera quantidade de parcelas não mudam as parcelas/somente se insere um novo item	Tamanho, Tipo de dados ou Domínio de validade de atributos
9	Cadastro Venda	Aceita venda sem item ou sem pagamento	Integridade Semântica
10	Data Vencimento(PG)	Aceita data menor que atual/Aceita data menor que data da venda	Tamanho, Tipo de dados ou Domínio de validade de atributos Dependência de atributos de tabelas diferentes
11	Quantidade de item	Não atualiza o estoque	Dependência de atributos de tabelas diferentes

Tabela 4.11 – Defeitos detectados na Tabela Produto

PRODUTO			
	Atributo Exercitado	**Casos de Testes com:**	**Tipo de Problema**
1	Codigo produto	código automático - ok	
2	CNPJ fornecedor	tamanho 11, menor que o normal	Tamanho, Tipo de dados ou Domínio de validade de atributos
3	Preço/Lucro/Estoque	quando tamanho maior mostra mensagem *oracle*	Tratar mensagem de exceção
4	Preço/Lucro/Estoque	Aceita valores negativos	Tamanho, Tipo de dados ou Domínio de validade de atributos

Estes critérios (intra-classe e inter-classe) também são eficazes na detecção de defeitos em comandos de consulta SQL (uso de *queries*) devido à integração de métodos e classes.

Há a situação em que classes não possuem nenhum elemento requerido para o teste de integração intra-classe, no entanto é exercitado no teste de integração inter-classe.

Existem também situações em que pode existir o ciclo 2 em diferentes classes, por exemplo, definir uma variável tabela *FOR* em um método da classe *Fornecedor.Java*, definir outra variável tabela *PRO* em um método da classe *Produto.Java* e usar a variável tabela *PRO* em um método da classe *Consultas.java*, como pode ser visualizado na Figura 4.7, existindo desta forma uma explosão de elementos requeridos, podendo ser detectados dependências. Não houve esta situação no teste em estudo de caso, mas convém citar a possibilidade desta ocorrência.

Como conclusão deste capítulo, pode-se verificar que algumas estratégias de teste força a detecção de erros de dependência de banco de dados, consistência de comandos de manipulação da SQL em relação a atributos das variáveis tabela, segurança das transações entre comandos de manipulação da base de dados. Se existir falta de dependências entre as tabelas, tanto o critério *todos-t-uso-ciclo1-intra* como o critério *todos-t-uso-ciclo2-intra* detectariam os erros, assim como os critérios *todos-t-uso-ciclo1-inter* e *todos-t-uso-ciclo2-inter*.

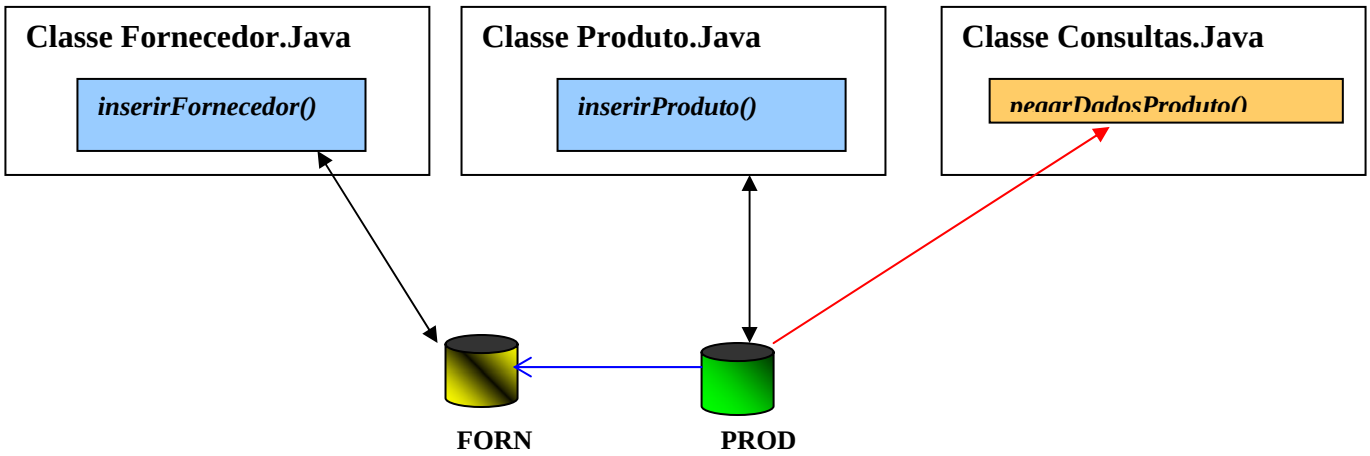


Figura 4.7 – Critério Inter-Classes-Ciclo2

4.5. Resultados obtidos dos testes realizados com a Ferramenta JaBUTi

Como comparação de abrangência entre os critérios de testes baseados na dependência dos dados (*todos-t-usos ciclo1* e *ciclo2*) para integração no Fluxo de Dados intra-classe, foi utilizado todos os casos de testes gerados para satisfazer os Elementos Requeridos deste critério numa nova etapa de teste, prevendo agora a cobertura dos elementos requeridos dos critérios implementados na Ferramenta JaBUTi (Todos-os-Potenciais-usos, Todos-os-usos, Todos-os-arcos e Todos-os-nós) visando a observar a cobertura adquirida para as classes em testes: Banco.Java, Operações.Java, CadastraClienteGUI.Java, CadastraVendaGUI.Java e Query.java. No início dessa etapa de teste, foi inicializado ("*startado*") o Banco de Dados para a utilização dos mesmos casos de teste dos critérios intra-classe ciclo1 e ciclo2. Nas Figuras de 4.8 a 4.11 são apresentados os resultados obtidos na Ferramenta JaBUTi com relação à cobertura dos elementos requeridos pelos critérios: *todos-os-nós*, *todos-os-arcos*, *todos-os-usos* e *todos-os-potenciais-usos*.

- Critério todos-os-nós:

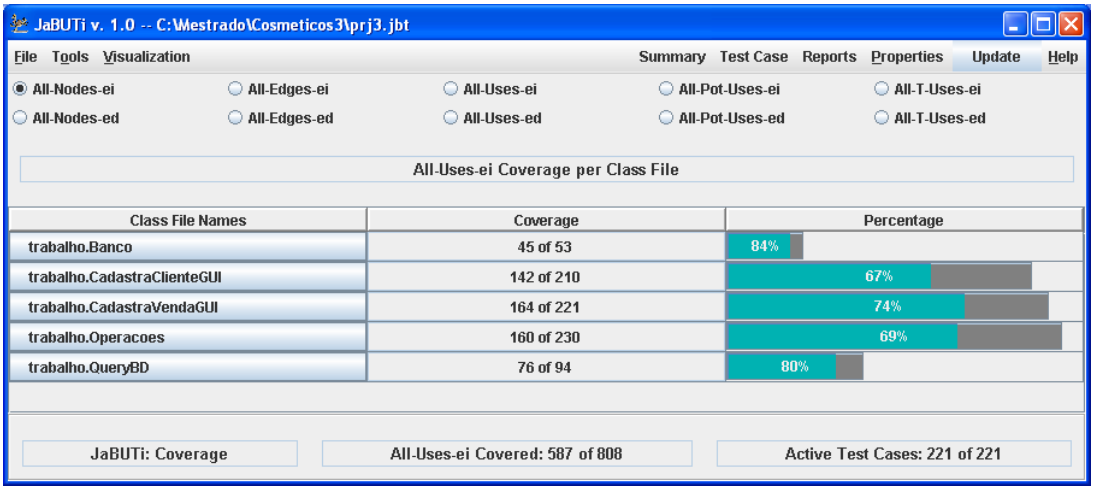


Figura 4.8 – Resultado dos testes obtidos pela Ferramenta JaBUTi, critério *todos-os-nós*

- Critério todos-os-arcos:

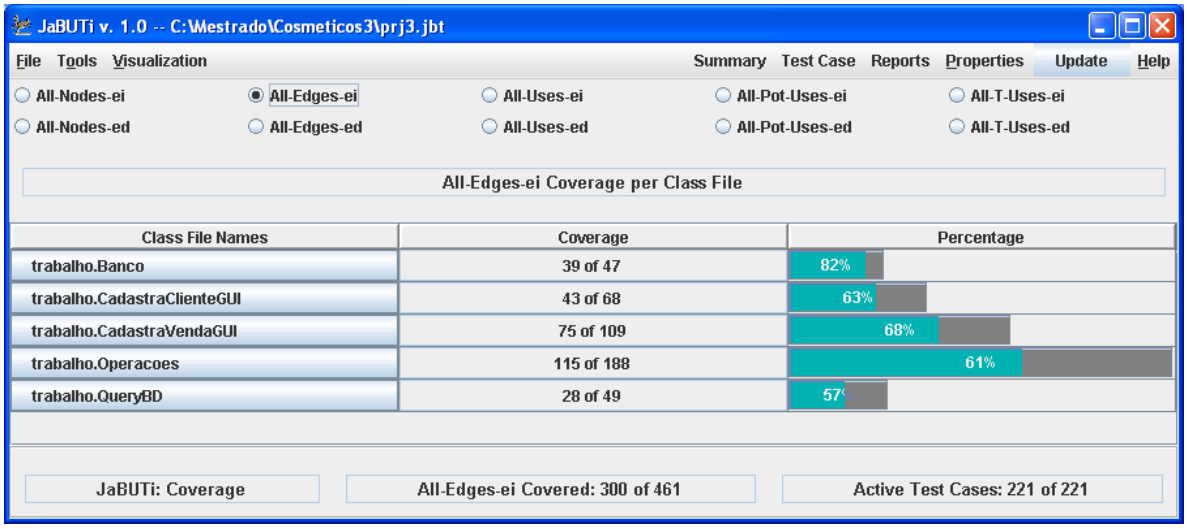


Figura 4.9 – Resultado dos testes obtidos pela Ferramenta JaBUTi, critério *todos-os-arcos*

- Critério todos-os-usos:

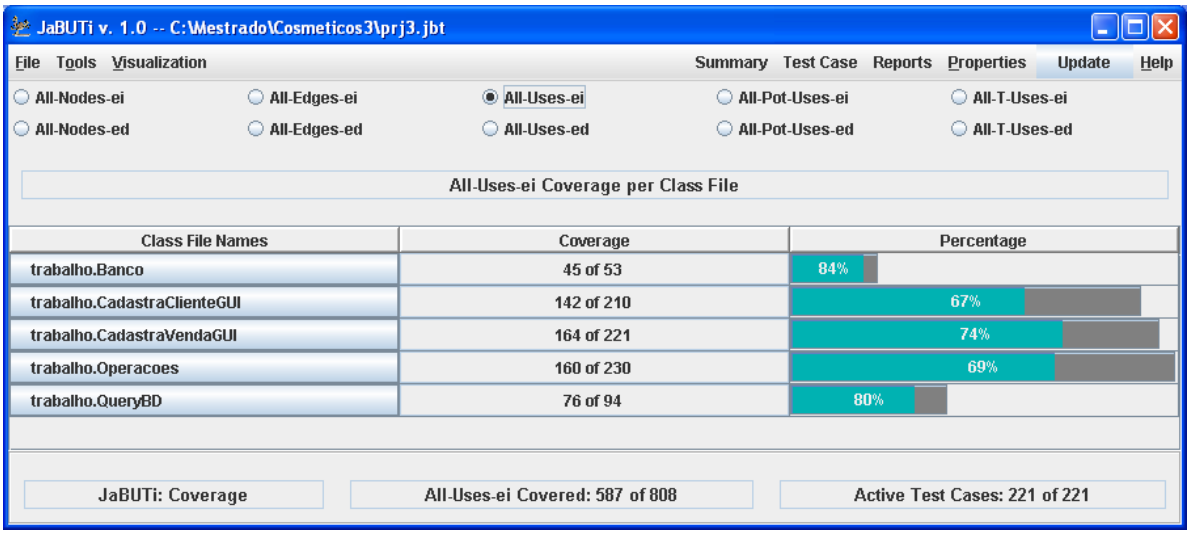


Figura 4.10 – Resultado dos testes obtidos pela Ferramenta JaBUTi, critério *todos-os-usos*

- Critério todos-os-potenciais-usos:

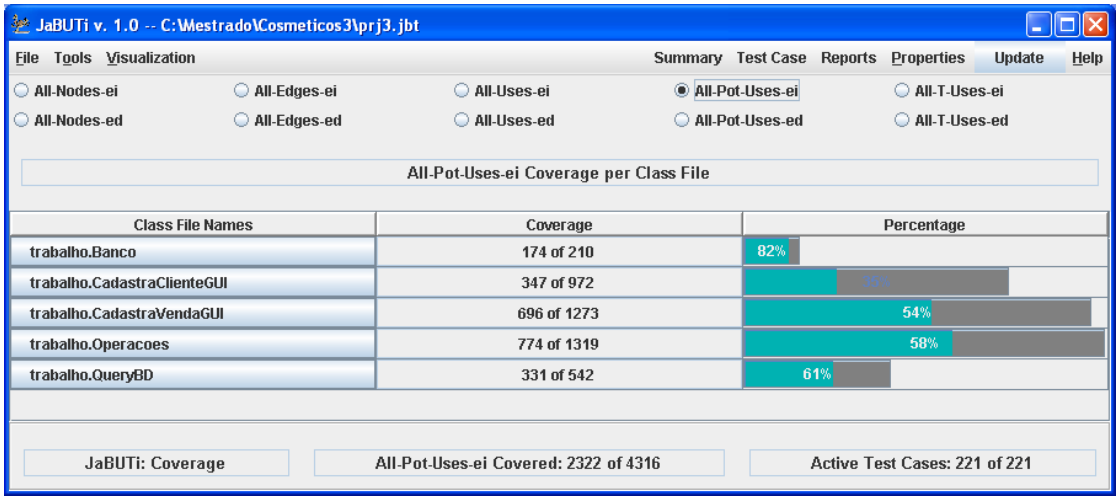


Figura 4.11 – Resultado dos testes obtidos pela Ferramenta JaBUTi, critério todos-os-potenciais-usos

Atualmente a Ferramenta JaBUTi, não realiza teste de integração. Podemos observar na Figura 4.8, que a cobertura para o critério todos-os-nós, foi de 84% para a classe Banco.java, 67% para a classe CadastraClienteGUI.java, 74% para a classe CadastraVendaGUI.java, 69% para a classe Operacoes.java e 80% para a classe QueryBD.java. Já na Figura 4.9, a cobertura para o critério todos-os-arcos foi de 82% para a classe Banco.java, 63% para a classe CadastraClienteGUI.java, 68% para a classe CadastraVendaGUI.java, 61% para a classe Operacoes.java e 57% para a classe QueryBD.java. Foi possível observar que os nós não satisfeitos pelos casos de testes extraídos dos critérios Todos-t-usos Ciclo1 e Ciclo2 (intra-classe) não possuem comandos de SQL, sendo assim mostra que os critérios são incomparáveis, porém são complementares por necessitar de ambas as análises (sendo que os critérios de ABDR são mais abrangentes por necessitarem da mesma *tupla* para satisfazer seus critérios). A mesma comparação pode ser realizada, com relação ao critério *todos-os-arcos* (resultado Figura 4.9), por se tratar de arcos que passam pelos mesmos nós não exercitados pelos casos de testes extraídos do teste de ABDR. E em alguns casos por se tratarem de arcos provenientes dos tratamentos de exceção

não exercitados nas etapas de testes de ABDR. Nas Figuras 4.12a e 4.12b são apresentados os Grafos dos métodos *inserirDados()* e *alterarDados()* respectivamente destacando os nós e arcos não cobertos pelos dados de teste nesta etapa (cor laranja e vermelho). Na Tabela 4.12 é mostrada uma relação de nós e arcos não executáveis pelos dados de testes extraídos da execução dos testes de ABDR para os principais métodos da classe Banco.Java.

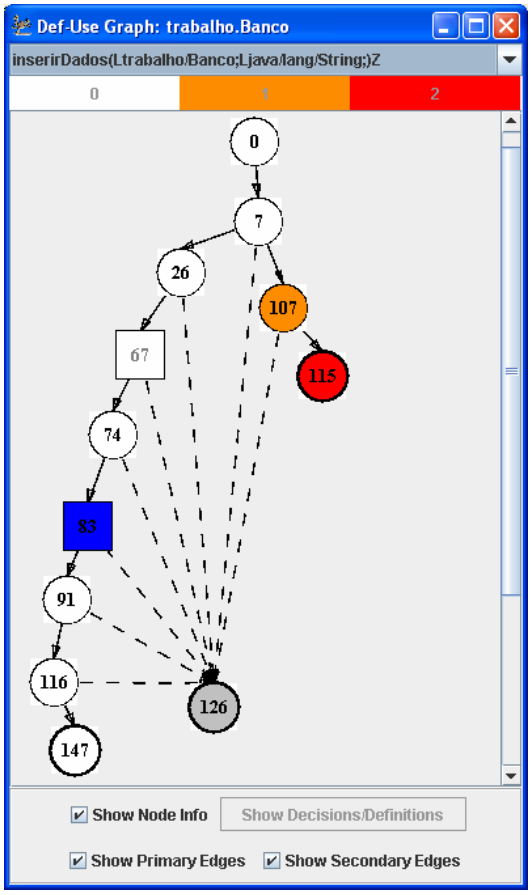


Figura 4.12a: *Def-use* Grafo(*inserirDados*).

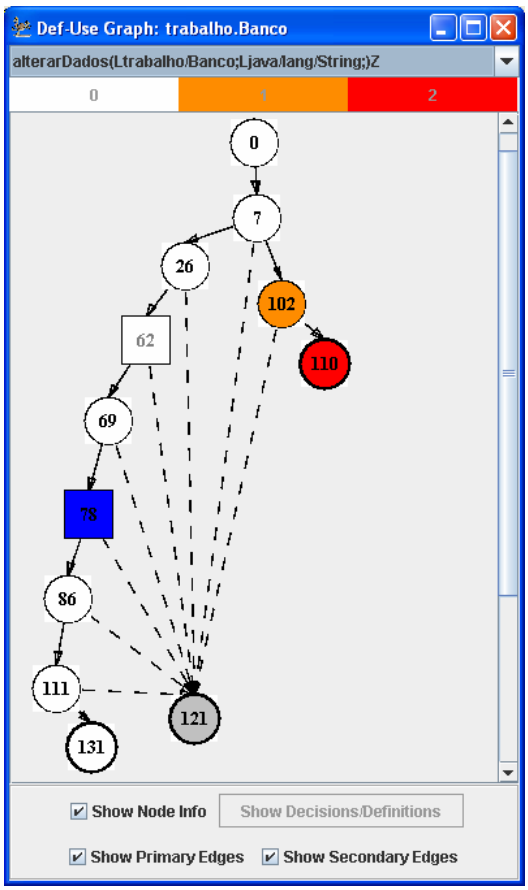


Figura 4.12b: *Def-use* Grafo(*alterarDados*).

Tabela 4.12 – Visualização dos elementos requeridos não executados para os critérios todos-os-nós e todos-os-arcos da Ferramenta JaBUTi

Método	**Nós (ne)**	**Arcos (ne)**
inserirDados	107 e 115	(7, 107) e (107, 115)
alterarDados	102 e 110	(7, 102) e (102, 110)
removerDados	124 e 132	(7, 124) e (124, 132)
verificarDados	117 e 125	(7, 117) e (117, 125)

A cobertura do critério todos-os-usos foi de 84% para a classe Banco.java, 67% para a classe CadastraClienteGUI.java, 74% para a classe CadastraVendaGUI.java, 69% para a classe Operacoes.java e 80% para a classe QueryBD.Java, na Figura 4.10. E, finalmente a cobertura para o critério todos-os-potenciais-usos foi de 82% para a classe Banco.java, 35% para a classe CadastraClienteGUI.java, 54% para a classe CadastraVendaGUI.java, 58% para a classe Operacoes.java e 61% para a classe QueryBD.java.

Foi observado que para a classe Banco.Java o total de elementos requeridos para o critério todos-os-nós é de 53; para o critério todos-os-arcos o total de elementos requeridos é 47; para o critério todos-os-usos o total de elementos requeridos é de 53 e para o critério todos-os-potenciais-usos é de 210. Foram encontrados 204 elementos requeridos para a os critérios intra-classe (ciclo 1 e ciclo 2), e foi observado que alguns casos de testes realizados, focando os critérios *todos-t-usos ciclo1* e *ciclo2*, cobriam vários elementos requeridos e para os critérios cobertos pela JaBUTi, não havia cobertura, pois, era realizado o mesmo caminho. Tendo em vista que para satisfazer as associações definição persistente e uso nos critérios de Spoto (2000), é exigida a mesma *tupla*, podendo passar pelo mesmo caminho (porém com a exigência de definir na primeira vez e de usar na segunda execução com a mesma *tupla*). Lembrando que a total cobertura do critério intra-classe (ciclo1 e ciclo 2) foi de 89,22%; sendo assim, verificamos de uma forma geral que os critérios não são comparáveis, por existir enfoques distintos de execução. Como observação, pode-se notar que executar inicialmente os critérios de ABDR torna o plano de teste mais criterioso com relação à geração dos casos de testes, tornando assim a escolha dos dados uma obrigação de se conhecer melhor o software e o banco de dados em teste.

5. CONCLUSÃO

Foi possível mostrar, através do experimento realizado neste trabalho que utilizou uma Aplicação em Linguagem Java com acesso ao SGBD Oracle9i e a Ferramenta JaBUTi no auxílio das etapas de testes para uma empresa de cosméticos, que a criação dos elementos requeridos baseados nos critérios de Spoto, intra-classe (ciclo1 e ciclo2) e inter-classe (ciclo1 e ciclo2) é um tanto trabalhoso, mas com a ajuda de uma ferramenta a tarefa de executar e avaliar os resultados torna-se mais branda. O planejamento dos testes tornou-se um veículo importante no auxílio da avaliação dos resultados (coberturas, estratégias de execução e obtenção dos dados de teste). Os critérios foram comparados e observados, e com isso mostrou-se que eles possuem exigências distintas por exercitarem aspectos diferentes e por serem complementares. Os programas utilizados neste experimento apresentaram muitas consistências de defesa, com relação à transação dos dados armazenados, constituindo-se em uma boa característica da aplicação.

Com os estudos realizados por Spoto (2000) que cria novos critérios de teste estrutural, explorando as relações de dados persistentes, focalizando as aplicações de banco de dados relacional, e com a realização deste trabalho foi possível mostrar que a eficácia do teste em banco de dados relacional pode ser adquirida quando executar os casos de testes, baseados nos critérios de ABDR, com valores que qualifiquem as principais características existentes nas associações *definição-t-uso* geradas pelos fluxos de dados intra-classe e inter-classe. Esses valores se relacionam com os diferentes tipos de integridades existentes (entidade, *tupla* e relacionamento). Bem como valores que possam acrescentar a visão de dependências entre atributos e entidades. Apesar de se utilizar a mesma *tupla* ao exercitar as definições e usos das tabelas, houve necessidade de se criar situações, na escolha dos dados, que provocassem a execução de comandos que levassem à detecção de defeitos, não protegidos pela Aplicação ou pelo próprio banco. Mostrando assim que estas situações apesar de não ocorrerem na prática

podem contribuir para detecção de falhas, verificando em que tipos de situações serão possíveis percebê-las.

Para as ABDRs, além dos critérios de teste baseados nas variáveis persistentes, criados por Spoto (2000), foram apresentadas algumas sugestões da escolha dos dados visando incluir as características de integridade referencial, integridade semântica, domínio, unicidade, dependência de atributos da mesma tabela, dependência de atributos de tabelas diferentes, o que contribuiu na melhoria da qualidade do teste.

O critério de teste *todos-t-uso-ciclo1-intra*, foi eficaz em detectar presença de dependências de tabelas, consistências de manipulação da SQL em relação a um atributo das variáveis tabela, consistências de segurança de transações. Assim como no critério *todos-t-uso-ciclo1-intra*, o critério de teste *todos-t-uso-ciclo2-intra*, também foi eficaz em detectar erros de consistências e segurança de transações, além de defeitos de dependências múltiplas, podendo ser aplicado em meio à necessidade de se definir uma tabela para conseguir a *associação-definição-t-uso* de outra variável tabela. De uma forma geral os critérios *todos-t-uso-ciclo1* e *todos-t-uso-ciclo2*, cobriram um percentual de 89,22% de um total de 204 elementos requeridos (Tabela 4.3) e o restante dos elementos requeridos não cobertos, são elementos não executáveis.

Os critérios de teste inter-classe, também são eficazes em detectar os mesmos defeitos dos critérios intra-classe, porém, a definição de uma variável tabela, está em uma classe e o uso desta está em outra classe. De um total de 200 elementos requeridos, os critérios inter-classe, cobriram um percentual de 51,22%, pois, alguns pontos das consultas SQLs da classe Operacoes.Java, não existia acesso ao usuário na escolha do dado, pois, eram consultas de consistência e interna ao programa.

Classes que não possuem pontos de definição persistente necessitaram dos critérios inter-classe para serem devidamente avaliados, mostrando que os critérios intra-classe e inter-

classe são complementares com relação a exercitar associações definição persistente com *t-uso* não cobertos na etapa de teste intra-classe. Isso ocorreu na classe Operacoes.java, que não possui comandos que caracterizam uma definição persistente de uma dada tabela ***t***, mas existem métodos nesta classe que usam tais tabelas; porém com os critérios inter-classe foi possível associar a classe Banco.java que possui a definição persistente da tabela ***t*** em seus métodos com a classe Operações.java possibilitando a geração e cobertura dos elementos requeridos inter-classe (Banco.Java X Operações.Java).

Para mostrar a eficácia do critério em relação à detecção de defeitos na Aplicação ou ausência de defesas na Aplicação foi acrescentada uma tabela com dependência de dados e com associações de integridade entre as tabelas em teste, sem que a Aplicação tenha alguma defesa (ou conhecimento de tal tabela). Com isso foi possível mostrar que os critérios intra-classe ou inter-classe geram os elementos requeridos provenientes dessas associações e obrigam que a execução exercite ou não tais defesas. Neste caso em particular, mostrou-se que os tratamentos de exceções foram exercitados e que tais tipos de erros são detectados pelo critério. Isso foi necessário devido às defesas implementadas na aplicação, não permitindo que tais defeitos ocorressem.

Posteriormente, utilizando os mesmos casos de teste, foram realizados os testes na Ferramenta Jabuti, para os critérios *todos-os-nós*, *todos-os-arcos*, *todos-os-usos* e *todos-os-potenciais-usos*, com as mesmas classes dos testes anteriores mais algumas classes de destaque no sistema. Os casos de testes gerados para os critérios de Spoto exigem uma estratégia que para satisfazer um dado elemento requerido é necessário executarmos uma definição persistente de uma variável tabela seguida de uma execução que caracteriza um uso da mesma tabela além de obrigar que ambas as execuções sejam na mesma *tupla*. Com isso houve uma necessidade de criarmos dezenas de casos de testes e observou-se que na visão dos critérios implementados na Ferramenta JaBUTi descritos acima, não havia necessidade de

utilizar as mesmas estratégias (execução aos pares definição e uso persistente e da mesma *tupla*), tornando assim várias execuções sem acréscimo na cobertura dos elementos requeridos dos critérios da JaBUTi. Porém, observou-se que os elementos requeridos nos métodos que utilizam os comandos de SQL obtiveram cobertura com a execução dos mesmos dados de testes (extraídos do teste para os critérios de SPOTO) e os métodos que não possuíam comandos de SQL não seguiam essa mesma permanência de cobertura. Isso mostra que os critérios implementados na JaBUTi e os critérios de Spoto são incomparáveis porém complementares.

Os critérios de integração *intra-modular* e *inter-modular*, os elementos requeridos, pedem o exercício de todas as possíveis combinações de definição persistente e usos nos comandos da SQL para cada variável tabela, sendo esse outro ponto que diferencia os critérios de Spoto dos demais critérios. Houveram elementos requeridos que não foram exercitados com a mesma *tupla* e, segundo Spoto (2000), esses elementos podem ser descartados após algumas tentativas, já que esses elementos forçam o testador a exercitar situações que podem revelar defeitos de implementação (exemplo: comando *Delete* - eliminar uma *tupla* que é chave estrangeira).

Os critérios de Spoto (2000) inicialmente criados foram baseados em Aplicações escritas em Linguagem C e no sistema em questão, as SQLs para definição e uso das variáveis persistentes as Unidades de Programas (métodos) eram separados para cada tabela. Nos testes realizados neste trabalho, na Linguagem Java, a definição e uso das variáveis tabela, concentram-se em duas classes: Banco.java e Operacoes.java; sendo que, para uma inserção, independente da tabela, era utilizado um mesmo método: *inserirDados(Banco bd, String query)*, em que apenas o parâmetro *query* indicava a variável tabela a ser definida ou utilizada. Desta forma, o caminho e o par *definição-t-uso* é o mesmo para todas as tabelas, no momento de inserir (uma característica diferente do teste de SPOTO), mudando apenas a

tabela através do parâmetro *query*. Desta forma, observa-se um aumento do fluxo de dados entre os métodos (devido às consistências de defesa desta aplicação), o que dificultou bastante a geração dos elementos requeridos e bem como a escolha dos casos de testes. Sendo assim, como trabalho futuro pretende-se estudar o acréscimo de um item de informação na criação da associação definição-persistente e uso de *t* acrescentando o ponto (nó ou comando) onde é construído o comando (*string* utilizado como parâmetro) que caracteriza a definição ou o uso persistente, aumentando assim os elementos requeridos e as chances de execuções.

Tendo em vista que a ABDR foi escrita em Java, esta possui classes com as seguintes características:

a. Classes que possuem os comandos de conexão ao banco de dados e execução de comandos DML (Classes Banco.Java e Operacoes.Java);

b. Classe que cria a *string* que representa o comando SQL (Classe Query.Java). Por exemplo: "INSERT INTO tabela_cliente VALUES ('" + cpf + "', '"+ rg + "', '" + nome + "', '" + end + "', '" + bair + "', '"+ tel + "', '" + cel + "', '" + nasc + "', '" + cid + "', '"+ est + "', '" + mail + "', '" + cep + "')"; e

c. Demais classes que fazem uso destas.

O ponto de uma definição persistente ocasionado pelo comando *INSERT*, por exemplo, é o mesmo para todas as tabelas, porém, a *string* que representa o comando *INSERT* é criada em uma outra classe e em métodos distintos dependendo da tabela em uso, dificultando, dessa forma, a etapa de geração dos elementos requeridos. Para contornar tais problemas foram incluídos na instrumentação do programa, pontos que identificam o caminho, a classe e o método que foi executado na criação do comando SQL (*string*) que identifica a definição persistente. Sugere-se como trabalhos futuros, que a Ferramenta gere elementos requeridos com a inclusão dos nós e respectivos métodos (e classes) responsáveis pela criação das *strings* de SQL para diferentes tabelas, visando assim tornar o teste mais

exigente. Exemplo: <CLI, (**12129, 0,** (13104, <62, 78>)), 13106, (62, 136)>. No exemplo o nó 0 do método 129 da classe 12000 é o ponto onde é gerada a ***string*** (***INSERT*** apresentado na característica ***b*** do parágrafo anterior).

Na Ferramenta Jabuti está sendo desenvolvido um módulo para incluir a implementação de teste de unidade do critério *todos-t-usos* (NARDI et. al., 2005). Como proposta de trabalho futuro espera-se que sejam desenvolvidos módulos para integração das unidades com a inclusão dos critérios de integração intra-classe e inter-classe na Ferramenta JaBUTi, visando assim auxiliar as etapas de instrumentação e geração de elementos requeridos, bem como análise de resultados (coberturas) em Aplicações escritas em linguagem Java. Também, como proposta de implementação futura, pode-se observar que os critérios de Spoto (2000) exercitam dependências de integridades entre tabelas (referencial *foreign key*) e de entidades (*primary key*), porém não existe nenhum elemento que gera uma avaliação com a integridade semântica; sendo assim, pode-se pensar em um critério de fluxo de dados que exercite tais dependências com base em atributos e não sobre a *tupla* especificamente.

Para o desenvolvimento deste trabalho, foram levantadas informações relativas ao teste estrutural de programas e ao teste estrutural de aplicações de banco de dados relacional. Foram estudados de forma abrangente o banco de dados *Oracle 9i* e a Ferramenta JaBUTi, pois, como ferramenta de apoio no estudo da eficácia do teste, foi fundamental para criação dos grafos para os critérios de Spoto, além de utilizá-la para comparação dos demais critérios já inseridos na ferramenta.

Os objetivos iniciais de estudar os vários critérios de teste estrutural, relacionados à ABDR, de verificar que um critério de teste exercita alguns defeitos associados a restrições de Banco de Dados Relacional resultou em sugestões na geração dos dados de testes, em acrescentar valores que avaliam alguns tipos de dependências ou domínios para detectar

defeitos em ABDR, e tais resultados apresentados mostraram que essa estratégia melhora o efeito dos resultados. Também foi mostrado que os critérios de Spoto forçam a detecção de erros de dependência de tabelas, a consistência de comandos DML, e a segurança das transações. Com isso, este trabalho atinge seus objetivos propostos inicialmente e abrem, dessa forma, novas perspectivas de estudos de Teste de ABDR.

6. REFERÊNCIAS

ARANHA, Maria C. L. F.M; MENDES, Nelson C; JINO, Mario; TOLEDO, Carlos M. T. **RDBTool: Uma Ferramenta de Apoio ao Teste de Bases de Dados Relacionais**.Anais. Curitiba: XI CITS, 2000.

BARBOSA, E.;VINCENZI, A.; MALDONADO, J.C. **Uma contribuição para Determinação de um Conjunto Essencial de Operadores de Mutação no Teste de Programas C**. Simpósio Brasileiro de Engenharia de Software – SBES'98, Anais. Maringá, Outubro 1998. p. 33-34.

BATISTA, Demerval Mendez. **DBValTool - Uma Ferramenta para Apoiar o Teste e a Validação de Projeto de Banco de Dados Relacional**. Dissertação de Mestrado. Orientador: Prof° Dr° Edmundo Sergio Spoto – Universidade Federal do Paraná, em convênio com o Departamento de Informática da Universidade Estadual de Maringá. Curitiba/PR, 2003.

BIO, Sergio R. **Sistemas de Informação: Um Enfoque Gerencial**. São Paulo: Atlas, 1996. 183p.

BASILI, R. **Quantitative Evaluation of Software Engineering Methodology**. Proceedings of the First Pan Pacific Computer Conference, Melbourne, Australia, Proceedings, Vol. 1, 1985. p. 379-398.

CHAIM, Marcos L. **POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados**. Orientador: Prof° Dr° Mario Jino. (Dissertação de Mestrado). Faculdade de Engenharia Elétrica da UNICAMP. Campinas/SP, 1991.

CHAYS, David; VOKOLOS, Filippos I.; WEYUKER, Elaine J. **A Framework for Testing Database Applications**. **ISSTA**. 00, ACM, Oregon, PO, 2000. p. 147-156

CHAYS, D.; Deng, Y., **"Demonstration of AGENDA Tool Set For Testing Relational Database Applications",** Proc. Of The Instl. Symposium on Software Engeneering, Portland, Oregon, 2003.

CHOW, T. S. **Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering**, v. 4, n. 3, mai. 1978. p. 178–187

CRESPO, Adalberto N, SILVA, Odair J., BORGES, Carlos A, SALVIANO, Clenio F., ARGOLLO, Miguel T.Jr., JINO, Mario. **Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo**. Simpósio Brasileiro de Qualidade de Software, 2004.

DATE, C.J**. Introdução a Sistemas de Banco de Dados**. 7ª ed. Rio de Janeiro: Campus, 2000.

DEMILLO, R.A., LIPTON, R.J, SAYWARD, F.G. **Hints on test data selection: Help for the practicing programmer**. **IEEE Computer**, v.11, n. 4, apr. 1978. p. 34-43

ELMASRI, Ramez and NAVATHE, Shankamt B. **Fundamentals of Database Systems.** Redwood City, CA. The Benjamin/Cummings Publishing Company, Inc. 2ª Edição, 1994.

FERREIRA, Aurélio Buarque de Hollanda. **Dicionário Aurélio básico da língua portuguesa.** Rio de Janeiro: Nova Fronteira, 1988, p. 687.

FREEDMAN, R. S. **Testability of software components**. **IEEE Transactions on Software Engineering,** v. 17, n. 6, jun. 1991. p. 553–564

HERMAN, P. M. **A data flow analysis approach to program testing**. **Australian Computer Journal**, v. 8, n. 3, nov. 1976. p. 92–96.

HORSTMANN, C.S. and CORNELL G. **"Core Java – Recursos Avançados".** vol. 2. São Paulo: Person Education do Brasil, 2001.

KITCHENHAM, Bárbara A; PFLEEGER, Shari Lawrence; PICKARD, Peter W. Jones; HOAGLIN, David C.; EMAN, Khaled El; ROSENBERG, Jarrett; **Preliminary**

Guidelines for Empirical Research in Software Engineering. **IEEE Transactions on Software Engineering,** v. 28, n. 8, aug. 2002. p. 721–734

KORTH, H.; SILBERSCHATZ, A. **Sistema de Banco de Dados**. Editora Makron Books, 3 ed., 1999

LEITÃO, P.S.J., VILELA, P.R.S., JINO, M., **"Mapping Faults to Failures in SQL Manipulation Commands".** ACS/IEEE Intl. Conference on Computer Systems and Applications, Cairo, Egypt, 3-6 January, 2005.

MALDONADO, José C.; CHAIN, Marcos L.; JINO, Mario. **Arquitetura de uma Ferramenta de Teste de Software de Apoio aos Critérios Potenciais Usos**. In: *Anais do XXII Congresso Nacional de Informática*. São Paulo, SP,1989.

NARDI, Paulo A.; DELAMARO, Marcio E.; SPOTO, Edmundo S.; VINCENZI, Auri M.R. **Utilização de Critérios Estruturais em Aplicações de Banco de Dados Java.** Seção de Ferramentas – SBES – Uberlância/MG, 2005.

MANNILA, H; RÄIHÄ, K.J. **Automatic Generation of Test Data for Relacional Queries. Journal of Computer and System Science**, vol.38, nº 2, 1989.

OFFUTT, A. Jefferson; PAN, Jie. **The dynamic domain reduction procedure for test data generation**.ISSE-TR-94-110, 1994.

ORACLE. **Developer's Guide and Reference.** Oracle Corporation, 1999.

OSTRAND, Thomas J. and BALCER, Marc J. **The Category-partition Method for Specifying and Generating Functional Tests. ACM** .Portland, Oregon, oct 19-20, 1987/1988.

PRESSMAN, R.S. **Engenharia de Software**.3.ed. São Paulo-SP: Makron Books, 1995.

RAPPS, S. and WEYUKER, E. J. **"Selection Software Test Data Using Data Flow Information", IEEE**, TSE, SE-11, April, 1985, p. 367-375.

SOMMERVILLE, Ian. **Engenharia de Software**. Tradução: Maurício de Andrade, São Paulo: Addison Wesley, 6ª edição, 2003.

SPOTO, Edmundo S. **Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional.** .(Tese). Campinas: UNICAMP-FEE-DCA, 2000.

SPOTO, Edmundo S., Jino, Mario, Maldonado, José C. **Teste Estrutural de Software: Uma abordagem para Aplicações de Banco de Dados Relacional**, 2003.

SPOTO, Edmundo S., Jino, Mario, Maldonado, José C. **Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional.** SBQS – p. 431-434 Porto Alegre/RS – PUCRS, 2005.

VILELA, P.R.S. **Critérios Potenciais Usos de Integração: Definição e Análise**. (Tese). Campinas FEEC – UNICAMP, 1998.

VINCENZI, Auri M.R. **Subsídios para o Estabelecimento de Estratégias de Teste Baseadas na Técnica de Mutação.** Orientador: Prof. Dr.José Carlos Maldonado. (Dissertação). ICMC/USP. São Carlos/SP, 1998.

VINCENZI, Auri M.R. **Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação**. Orientador: Prof° Dr°José Carlos Maldonado. Co-Orientador: Prof° Dr° Márcio Eduardo Delamaro. (Tese). USP. São Carlos/SP, 2004.

VINCENZI, Auri M.R., MALDONADO, José C., DELAMARO, Marcio E., SPOTO, Edmundo S., WONG, Eric. **Capítulo 1 Software Baseado em Componentes: Uma Revisão sobre Teste**. vol. 2693 de **Lecture Notes in Computer Science**, cap. Component-Basead Software: An Overview of Testing, New York, NY:Springer-Verlag, 2003.

VINCENZI, Auri M.R. et al. **JaBUTi – Java Bytecode Understanding and Testing**. São Carlos, SP: Universidade de São Paulo – UPS, 2003.

WERNER, Claudete, **Avaliação Experimental do Critério de Teste Mutação Dual.** Orientador Prof° Dr° Márcio Eduardo Delamaro. (Dissertação de Mestrado). Fundação de Ensino Eurípedes Soares da Rocha, FEESR. Marília/SP, 2005

WU, Xintao; WANG, Yongge; ZHENG, Yuliang. **"Privacy Preserving Database Application Testing",** ACM – WPES'03, October, 2003 – Washington – USA.

ANEXO I - Elementos Requeridos Intra-Classe e Inter-Classe

Para melhorar a visualização dos elementos requeridos, iremos abreviar os nomes das tabelas da seguinte forma:
TABELA_CLIENTE = CLI
TABELA_FUNCIONARIO = FUNC
TABELA_FORNECEDOR = FORN
TABELA_PRODUTO = PROD
TABELA_VENDA = VEND
TABELA_COMPRA = COMP
TABELA_ITENS = ITEM
TABELA_PAGAMENTO = PAG

Elementos Requeridos para o critério Intra-Classe

Num	Elementos Requeridos
	Inclusão Clientes
1	<CLI, (13104, <62, 78>), 13104, (62, 78)> {Insert/Insert}
2	<CLI, (13104, <62, 78>), 13104, (62, 121)> {Insert/Insert catch}
3	<CLI, (13104, <62, 78>), 13105, (62, 78)> {Insert/Update}
4	<CLI, (13104, <62, 78>), 13105, (62, 121)> {Insert/Update catch}
5	<CLI, (13104, <62, 78>), 13106, (62, 100)> {Insert/Delete }
6	**<CLI, (13104, <62, 78>), 13106, (62, 136)> {Insert/Delete catch}**
7	<CLI, (13104, <62, 78>), 13108, (62, 69)> {Insert/Select }
8	<CLI, (13104, <62, 78>), 13108, (62, 136)> {Insert/Select catch }
9	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 69)> {insert/select}
10	**<CLI, (13104, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 136)>** {insert/select catch}
11	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 69)>
12	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 136)>
13	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69
14	**<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	Alteração Clientes
15	<CLI, (13105, <62, 78>), 13104, (62, 78)> {Update/Insert}
16	<CLI, (13105, <62, 78>), 13104, (62, 121)> {Update/Insert catch}
17	<CLI, (13105, <62, 78>), 13105, (62, 78)> {Update/Update }
18	<CLI, (13105, <62, 78>), 13105, (62, 121)> {Update/Update catch}
19	<CLI, (13105, <62, 78>), 13106, (62, 100)> {Update/Delete }
20	**<CLI, (13105, <62, 78>), 13106, (62, 136)> {Update/Delete catch}**
21	<CLI, (13105, <62, 78>), 13108, (62, 69)> {Update/Select }
22	<CLI, (13105, <62, 78>), 13108, (62, 136)> {Update/Select }
23	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 69)> {Update/select }
24	**<CLI, (13105, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 136)> {Update/select }**
25	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 69)>
26	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 136)>
27	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
28	**<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	Exclusão Clientes

29	<CLI, (13106, <62, 100>), 13104, (62, 78)> { Delete/Insert}
30	<CLI, (13106, <62, 100>), 13104, (62, 121)> { Delete /Insert catch}
31	<CLI, (13106, <62, 100>), 13105, (62, 78)> { Delete /Update }
32	<CLI, (13106, <62, 100>), 13105, (62, 121)> { Delete /Update catch}
33	<CLI, (13106, <62, 100>), 13106, (62, 100)> { Delete /Delete }
34	**<CLI, (13106, <62, 100>), 13106, (62, 136)> { Delete /Delete catch }**
35	<CLI, (13106, <62, 100>), 13108, (62, 69)> { Delete /Select }
36	<CLI, (13106, <62, 100>), 13108, (62, 136)> { Delete /Select catch }
37	<CLI, (13106, <62, 100>), {VEND, CLI, FUNC}, 13108, (62, 69)>
38	**<CLI, (13106, <62, 100>), {VEND, CLI, FUNC}, 13108, (62, 136)>**
39	<CLI, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 13108, (62, 69)>
40	<CLI, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 13108, (62, 136)>
41	<CLI, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
42	**<CLI, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	TABELA FUNCIONARIO
	Inclusão Funcionário
43	<FUNC, (13104, <62, 78>), 13104, (62, 78)>
44	<FUNC, (13104, <62, 78>), 13104, (62, 121)>
45	<FUNC, (13104, <62, 78>), 13105, (62, 78)>
46	<FUNC, (13104, <62, 78>), 13105, (62, 121)>
47	<FUNC, (13104, <62, 78>), 13106, (62, 100)>
48	<FUNC, (13104, <62, 78>), 13106, (62, 136)>
49	<FUNC, (13104, <62, 78>), 13108, (62, 69)>
50	<FUNC, (13104, <62, 78>), 13108, (62, 136)>
51	<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 69)>
52	**<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 136)>**
53	<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 69)>
54	<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 136)>
55	<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
56	**<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	Alteração Funcionário
57	<FUNC, (13105, <62, 78>), 13104, (62, 78)>
58	<FUNC, (13105, <62, 78>), 13104, (62, 121)>
59	<FUNC, (13105, <62, 78>), 13105, (62, 78)>
60	<FUNC, (13105, <62, 78>), 13105, (62, 121)>
61	<FUNC, (13105, <62, 78>), 13106, (62, 100)>
62	<FUNC, (13105, <62, 78>), 13106, (62, 136)>
63	<FUNC, (13105, <62, 78>), 13108, (62, 69)>
64	<FUNC, (13105, <62, 78>), 13108, (62, 136)>
65	<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 69)>
66	**<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC}, 13108, (62, 136)>**
67	<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 69)>
68	<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 13108, (62, 136)>
69	<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
70	**<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	Exclusão Funcionário
71	<FUNC, (13106, <62, 100>), 13104, (62, 78)>
72	<FUNC, (13106, <62, 100>), 13104, (62, 121)>

73	<FUNC, (13106, <62, 100>), 13105, (62, 78)>
74	<FUNC, (13106, <62, 100>), 13105, (62, 121)>
75	<FUNC, (13106, <62, 100>), 13106, (62, 100)>
76	<FUNC, (13106, <62, 100>), 13106, (62, 136)>
77	<FUNC, (13106, <62, 100>), 13108, (62, 69)>
78	<FUNC, (13106, <62, 100>), 13108, (62, 136)>
79	<FUNC, (13106, <62, 100>), {VEND, CLI, FUNC}, 13108, (62, 69)>
80	**<FUNC, (13106, <62, 100>), {VEND, CLI, FUNC}, 13108, (62, 136)>**
81	<FUNC, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 13108, (62, 69)>
82	<FUNC, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 13108, (62, 136)>
83	<FUNC, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
84	**<FUNC, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	TABELA FORNECEDOR
	Inclusão Fornecedor
85	<FORN, (13104, <62, 78>), 13104, (62, 78)>.
86	<FORN, (13104, <62, 78>), 13104, (62, 121)>
87	<FORN, (13104, <62, 78>), 13105, (62, 78)>
88	<FORN, (13104, <62, 78>), 13105, (62, 121)>
89	<FORN, (13104, <62, 78>), 13106, (62, 100)>
90	**<FORN, (13104, <62, 78>), 13106, (62, 136)>**
91	<FORN, (13104, <62, 78>), 13108, (62, 69)>
92	<FORN, (13104, <62, 78>), 13108, (62, 136)>
93	<FORN, (13104, <62, 78>), {FORN, PROD}, (13108, (62, 69))>
94	<FORN, (13104, <62, 78>), {FORN, PROD}, (13108, (62, 136))>
	Alteração Fornecedor
95	<FORN, (13105, <62, 78>), 13104, (62, 78)>.
96	<FORN, (13105, <62, 78>), 13104, (62, 121)>
97	<FORN, (13105, <62, 78>), 13105, (62, 78)>
98	<FORN, (13105, <62, 78>), 13105, (62, 121)>
99	<FORN, (13105, <62, 78>), 13106, (62, 100)>
100	**<FORN, (13105, <62, 78>), 13106, (62, 136)>**
101	<FORN, (13105, <62, 78>), 13108, (62, 69)>
102	<FORN, (13105, <62, 78>), 13108, (62, 136)>
103	<FORN, (13105, <62, 78>), {FORN, PROD}, (13108, (62, 69))>
104	<FORN, (13105, <62, 78>), {FORN, PROD}, (13108, (62, 136))>
	Exclusão Fornecedor
105	<FORN, (13106, <62, 100>), 13104, (62, 78)>.
106	<FORN, (13106, <62, 100>), 13104, (62, 121)>
107	<FORN, (13106, <62, 100>), 13105, (62, 78)>
108	<FORN, (13106, <62, 100>), 13105, (62, 121)>
109	<FORN, (13106, <62, 100>), 13106, (62, 100)>
110	**<FORN, (13106, <62, 100>), 13106, (62, 136)>**
111	<FORN, (13106, <62, 100>), 13108, (62, 69)>
112	<FORN, (13106, <62, 100>), 13108, (62, 136)>
113	<FORN, (13106, <62, 100>), {FORN, PROD}, (13108, (62, 69))>
114	<FORN, (13106, <62, 100>), {FORN, PROD}, (13108, (62, 136))>

	TABELA PRODUTO (dependência com tabela fornecedor)- Cód.Prod. Automático
	Inclusão Produto
115	<PROD, (13104, <62, 78>), PROD, (13104, (62, 78))>
116	<PROD, (13104, <62, 78>), PROD, (13104, (62, 121))>
117	<PROD, (13104, <62, 78>), PROD, (13105, (62, 78))>
118	<PROD, (13104, <62, 78>), PROD, (13105, (62, 121))>
119	<PROD, (13104, <62, 78>), PROD, (13106, (62, 100))>
120	**<PROD, (13104, <62, 78>), PROD, (13106, (62, 136))>**
121	<PROD, (13104, <62, 78>), {FORN, PROD}, (13108, (62, 69))>
122	<PROD, (13104, <62, 78>), {FORN, PROD}, (13108, (62, 136))>
	Alteração Produto
123	<PROD, (13105, <62, 78>), PROD, (13104, (62, 78))>
124	<PROD, (13105, <62, 78>), PROD, (13104, (62, 121))>
125	<PROD, (13105, <62, 78>), PROD, (13105, (62, 78))>
126	<PROD, (13105, <62, 78>), PROD, (13105, (62, 121))>
127	<PROD, (13105, <62, 78>), PROD, (13106, (62, 100))>
128	**<PROD, (13105, <62, 78>), PROD, (13106, (62, 136))>**
129	<PROD, (13105, <62, 78>), {FORN, PROD}, (13108, (62, 69))>
130	<PROD, (13105, <62, 78>), {FORN, PROD}, (13108, (62, 136))>
	Exclusão Produto
131	<PROD, (13106, <62, 100>), PROD, (13104, (62, 78))>
132	<PROD, (13106, <62, 100>), PROD, (13104, (62, 121))>
133	<PROD, (13106, <62, 100>), PROD, (13105, (62, 78))>
134	<PROD, (13106, <62, 100>), PROD, (13105, (62, 121))>
135	<PROD, (13106, <62, 100>), PROD, (13106, (62, 100))>
136	**<PROD, (13106, <62, 100>), PROD, (13106, (62, 136))>**
137	**<PROD, (13106, <62, 100>), {FORN, PROD}, (13108, (62, 69))>**
138	**<PROD, (13106, <62, 100>), {FORN, PROD}, (13108, (62, 136))>**
	Inclusão Fornecedor/Inclusão Produto
139	<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (13104, (62, 78))>
140	<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (13104, (62, 121))>
141	<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (13105, (62, 78))>
142	<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (13105, (62, 121))>
143	<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (13106, (62, 100))>
144	**<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (13106, (62, 136))>**
145	<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {FORN, PROD}, (13108, (62, 69))>
146	<FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {FORN, PROD}, (13108, (62, 136))>
	TABELA VENDA (dependência com tabela Cliente e Funcionário)
	OBS: no sistema não tem opção de alteração de vendas
	Inclusão Venda
147	<VEND, (13104, <62, 78>), VEND, (13104, (62, 78))>
148	<VEND, (13104, <62, 78>), VEND, (13104, (62, 121))>
149	<VEND, (13104, <62, 78>), VEND, (13106, (62, 100))>
150	**<VEND, (13104, <62, 78>), VEND, (13106, (62, 136))>**
151	<VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (13108, (62, 69))>
152	**<VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (13108, (62, 136))>**
153	<VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (13108, (62, 69))>

154	<VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (13108, (62, 136))>
155	<VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
156	**<VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	PARA ESTE PROGRAMA, VENDAS NÃO EXISTE ALTERAÇÃO
	Exclusão Venda
157	<VEND, (13106, <62, 100>), VEND, (13104, (62, 78))>
158	<VEND, (13106, <62, 100>), VEND, (13104, (62, 121))>
159	<VEND, (13106, <62, 100>), VEND, (13106, (62, 100))>
160	**<VEND, (13106, <62, 100>), VEND, (13106, (62, 136))>**
161	<VEND, (13106, <62, 100>), {VEND, CLI, FUNC}, (13108, (62, 69))>
162	**<VEND, (13106, <62, 100>), {VEND, CLI, FUNC}, (13108, (62, 136))>**
163	<VEND, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (13108, (62, 69))>
164	<VEND, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (13108, (62, 136))>
165	<VEND, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
166	**<VEND, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	Inclusão Cliente/Inclusão Venda
167	<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), VEND, (13104, (62, 78))>
168	<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), VEND, (13104, (62, 121))>
169	<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), VEND, (13106, (62, 100))>
170	**<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), VEND, (13106, (62, 136))>**
171	<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (13108, (62, 69))>
172	**<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (13108, (62, 136))>**
173	<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (13108, (62, 69))>
174	311. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (13108, (62, 136))>
175	<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
176	**<CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	TABELA ITENS (dependência com tabela Venda e Produto)
177	<VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), ITEM, (13104, (62, 78))>
178	<VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), ITEM, (13104, (62, 121))>
179	<VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), ITEM, (13106, (62, 100))>
180	**<VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), ITEM, (13106, (62, 136))>**
181	<VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
182	**<VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	Exclusão Item
183	<ITEM, (13106, <62, 100>), ITEM, (13104, (62, 78))>
184	<ITEM, (13106, <62, 100>), ITEM, (13104, (62, 121))>
185	<ITEM, (13106, <62, 100>), ITEM, (13106, (62, 100))>
186	**<ITEM, (13106, <62, 100>), ITEM, (13106, (62, 136))>**
187	<ITEM, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69
188	**<ITEM, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	TABELA PAGAMENTO (dependência com tabela Venda)
	OBS: a tabela pagamento é definida juntamente com tabela venda

189	<VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (13104, (62, 78))>
190	<VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (13104, (62, 121))>
191	<VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (13106, (62, 100))>
192	**<VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (13106, (62, 136))>**
193	<VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (13108, (62, 69))>
194	<VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (13108, (62, 136))>
195	<VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
196	**<VEND, (13104, <62, 78>), <PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**
	Exclusão Pagamento
197	<PAG, (13106, <62, 100>), PAG, (13104, (62, 78))>
198	<PAG, (13106, <62, 100>), PAG, (13104, (62, 121))>
199	<PAG, (13106, <62, 100>), PAG, (13106, (62, 100))>
200	**<PAG, (13106, <62, 100>), PAG, (13106, (62, 136))>**
201	<PAG, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (13108, (62, 69))>
202	<PAG, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (13108, (62, 136))>
203	<PAG, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 69))>
204	**<PAG, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (13108, (62, 136))>**

Elementos Requeridos para o critério Inter-Classe

Num	Elementos Requeridos
	TABELA CLIENTES
	Inclusão Clientes
1	<CLI, (13104, <62, 78>), 11105, (70, 77)> {Insert/Select cliente}
2	**<CLI, (13104, <62, 78>), 11105, (70, 266)> {Insert/Select cliente catch}**
3	<CLI, (13104, <62, 78>), 11119, (75, 83)> {Insert/select p/ montar grid}
4	**<CLI, (13104, <62, 78>), 11119, (75, 232)> {Insert/select p/ montar grid catch}**
5	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 83)>
6	**<CLI, (13104, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 232)>**
7	<CLI, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 77)>
8	**<CLI, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 329)>**
9	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 83)>
10	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 232)>
11	<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
12	**<CLI, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Alteração Clientes
13	<CLI, (13105, <62, 78>), 11105, (70, 77)>
14	**<CLI, (13105, <62, 78>), 11105, (70, 266)>**
15	<CLI, (13105, <62, 78>), 11119, (75, 83)>
16	**<CLI, (13105, <62, 78>), 11119, (75, 232)>**
17	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 83)>
18	**<CLI, (13105, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 232)>**
19	<CLI, (13105, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 77)>
20	**<CLI, (13105, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 329)>**

21	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 83)>
22	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 232)>
23	<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
24	**<CLI, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Exclusão Clientes
25	<CLI, (13106, <62, 100>), 11105, (70, 77)>
26	**<CLI, (13106, <62, 100>), 11105, (70, 266)>**
27	<CLI, (13106, <62, 100>), 11119, (75, 83)>
28	**<CLI, (13106, <62, 100>), 11119, (75, 232)>**
29	<CLI, (13106, <62, 100>), {VEND, CLI, FUNC}, 11119, (75, 83)>
30	**<CLI, (13106, <62, 100>), {VEND, CLI, FUNC}, 11119, (75, 232)>**
31	<CLI, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 77)>
32	**<CLI, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 329)>**
33	<CLI, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 11119, (75, 83)>
34	**<CLI, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 11119, (75, 232)>**
35	<CLI, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
36	**<CLI, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Inclusão Funcionário
37	<FUNC, (13104, <62, 78>), 11106, (70, 77)>
38	**<FUNC, (13104, <62, 78>), 11106, (70, 393)>**
39	<FUNC, (13104, <62, 78>), 11106, (296, 303)>
40	**<FUNC, (13104, <62, 78>), 11106, (296, 393)>**
41	<FUNC, (13104, <62, 78>), 11119, (75, 83)>
42	**<FUNC, (13104, <62, 78>), 11119, (75, 232)>**
43	<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 83)>
44	**<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 232)>**
45	<FUNC, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 77)>
46	**<FUNC, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 329)>**
47	<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 83)>
48	**<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 232)>**
49	<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
50	**<FUNC, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Alteração Funcionário
51	<FUNC, (13105, <62, 78>), 11106, (70, 77)>
52	**<FUNC, (13105, <62, 78>), 11106, (70, 393)>**
53	<FUNC, (13105, <62, 78>), 11106, (296, 303)>
54	**<FUNC, (13105, <62, 78>), 11106, (296, 393)>**
55	<FUNC, (13105, <62, 78>), 11119, (75, 83)>
56	**<FUNC, (13105, <62, 78>), 11119, (75, 232)>**
57	<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 83)>
58	**<FUNC, (13105, <62, 78>), {VEND, CLI, FUNC}, 11119, (75, 232)>**
59	109. <FUNC, (13105, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 77)>
60	**110. <FUNC, (13105, <62, 78>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 329)>**
61	111. <FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 83)>
62	**112. <FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG}, 11119, (75, 232)>**
63	388. <FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
64	**389. <FUNC, (13105, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**

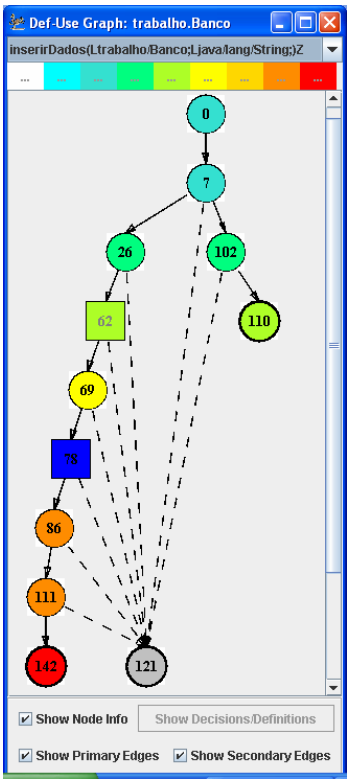
	Exclusão Funcionário
65	123. <FUNC, (13106, <62, 100>), 11106, (70, 77)>
66	**124. <FUNC, (13106, <62, 100>), 11106, (70, 393)>**
67	125. <FUNC, (13106, <62, 100>), 11106, (296, 303)>
68	**126. <FUNC, (13106, <62, 100>), 11106, (296, 393)>**
69	127. <FUNC, (13106, <62, 100>), 11119, (75, 83)>
70	**128. <FUNC, (13106, <62, 100>), 11119, (75, 232)>**
71	131. <FUNC, (13106, <62, 100>), {VEND, CLI, FUNC}, 11119, (75, 83)>
72	**132. <FUNC, (13106, <62, 100>), {VEND, CLI, FUNC}, 11119, (75, 232)>**
73	133. <FUNC, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 77)>
74	**134. <FUNC, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, 11110, (70, 329)> {Del/sel p/ montar grid}**
75	135. <FUNC, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 11119, (75, 83)>
76	**136. <FUNC, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, 11119, (75, 232)>**
77	392. <FUNC, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
78	**393. <FUNC, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	TABELA FORNECEDOR
	Inclusão Fornecedor
79	147. <FORN, (13104, <62, 78>), 11107, (70, 77)>
80	**148. <FORN, (13104, <62, 78>), 11107, (70, 233)>**
81	149. <FORN, (13104, <62, 78>), 11119, (75, 83)>
82	**150. <FORN, (13104, <62, 78>), 11119, (75, 232)>**
83	153. <FORN, (13104, <62, 78>), {FORN, PROD}, (11119, (75, 83))>
84	**154. <FORN, (13104, <62, 78>), {FORN, PROD}, (11119, (75, 232))>**
85	155. <FORN, (13104, <62, 78>), {FORN, PROD}, (11108, (70, 77))>
86	**156. <FORN, (13104, <62, 78>), {FORN, PROD}, (11108, (70, 244))>**
	Alteração Fornecedor
87	165. <FORN, (13105, <62, 78>), 11107, (70, 77)>
88	**166. <FORN, (13105, <62, 78>), 11107, (70, 233)>**
89	167. <FORN, (13105, <62, 78>), 11119, (75, 83)>
90	**168. <FORN, (13105, <62, 78>), 11119, (75, 232)>**
91	171. <FORN, (13105, <62, 78>), {FORN, PROD}, (11119, (75, 83))>
92	**172. <FORN, (13105, <62, 78>), {FORN, PROD}, (11119, (75, 232))>**
93	173. <FORN, (13105, <62, 78>), {FORN, PROD}, (11108, (70, 77))>
94	**174. <FORN, (13105, <62, 78>), {FORN, PROD}, (11108, (70, 244))>**
	Exclusão Fornecedor
95	183. <FORN, (13106, <62, 100>), 11107, (70, 77)>
96	**184. <FORN, (13106, <62, 100>), 11107, (70, 233)>**
97	185. <FORN, (13106, <62, 100>), 11119, (75, 83)>
98	**186. <FORN, (13106, <62, 100>), 11119, (75, 232)>**
99	189. <FORN, (13106, <62, 100>), {FORN, PROD}, (11119, (75, 83))>
100	**190. <FORN, (13106, <62, 100>), {FORN, PROD}, (11119, (75, 232))>**
101	191. <FORN, (13106, <62, 100>), {FORN, PROD}, (11108, (70, 77))>
102	**192. <FORN, (13106, <62, 100>), {FORN, PROD}, (11108, (70, 244))>**
	TABELA PRODUTO (Dependência Fornecedor - Cód.pro automático)
	Inclusão Produto
103	203. <PROD, (13104, <62, 78>), PROD, (11114, (70, 77))>
104	**204. <PROD, (13104, <62, 78>), PROD, (11114, (70, 156))>**

105	205. <PROD, (13104, <62, 78>), {ITEM, PROD}, (11112, (74, 81))>
106	**206. <PROD, (13104, <62, 78>), {ITEM, PROD}, (11112, (74, 228))>**
107	209. <PROD, (13104, <62, 78>), {FORN, PROD}, (11119, (75, 83))>
108	**210. <PROD, (13104, <62, 78>), {FORN, PROD}, (11119, (75, 232))>**
109	211. <PROD, (13104, <62, 78>), {FORN, PROD}, (11108, (70, 77))>
110	**212. <PROD, (13104, <62, 78>), {FORN, PROD}, (11108, (70, 244))>**
	Alteração Produto
111	223. <PROD, (13105, <62, 78>), PROD, (11114, (70, 77))>
112	**224. <PROD, (13105, <62, 78>), PROD, (11114, (70, 156))>**
113	225. <PROD, (13105, <62, 78>), {ITEM, PROD}, (11112, (74, 81))>
114	**226. <PROD, (13105, <62, 78>), {ITEM, PROD}, (11112, (74, 228))>**
115	229. <PROD, (13105, <62, 78>), {FORN, PROD}, (11119, (75, 83))>
116	**230. <PROD, (13105, <62, 78>), {FORN, PROD}, (11119, (75, 232))>**
117	231. <PROD, (13105, <62, 78>), {FORN, PROD}, (11108, (70, 77))>
118	**232. <PROD, (13105, <62, 78>), {FORN, PROD}, (11108, (70, 244))>**
	Exclusão Produto
119	241. <PROD, (13106, <62, 100>), PROD, (11114, (70, 77))>
120	**242. <PROD, (13106, <62, 100>), PROD, (11114, (70, 156))>**
121	247. <PROD, (13106, <62, 100>), {ITEM, PROD}, (11112, (74, 81))>
122	**248. <PROD, (13106, <62, 100>), {ITEM, PROD}, (11112, (74, 228))>**
123	243. <PROD, (13106, <62, 100>), {PROD, FORN}, (11119, (75, 83))>
124	**244. <PROD, (13106, <62, 100>), {PROD, FORN}, (11119, (75, 232))>**
125	245. <PROD, (13106, <62, 100>), {PROD, FORN}, (11108, (70, 77))>
126	**246. <PROD, (13106, <62, 100>), {PROD, FORN}, (11108, (70, 244))>**
	Inclusão Fornecedor/Inclusão Produto
127	267. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (11114, (70, 77))>
128	**268. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), PROD, (11114, (70, 156))>**
129	263. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {ITEM, PROD}, (11112, (74, 81))>
130	**264. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {ITEM, PROD}, (11112, (74, 228))>**
131	259. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {PROD, FORN}, (11119, (75, 83))> {Ins forn/i}
132	**260. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {PROD, FORN}, (11119, (75, 232))>**
133	261. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {PROD, FORN}, (11108, (70, 77))>
134	**262. <FORN, (13104, <62, 78>), PROD, (13104, <62, 78>), {PROD, FORN}, (11108, (70, 244))>**
	TABELA VENDA (Dependência com Tabela Cliente e Funcionário)
	OBS: no sistema não tem opção de alteração de vendas
	Inclusão Venda
135	274. <VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (11119, (75, 83))>
136	**275. <VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (11119, (75, 232))>**
137	276. <VEND, (13104, <62, 78>), VEND, (11116, (70, 77))>
138	**277. <VEND, (13104, <62, 78>), VEND, (11116, (70, 156))>**
139	<VEND, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 77))>
140	**<VEND, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 329))>**
141	280. <VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (11119, (75, 83))>
142	**281. <VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (11119, (75, 232))>**

143	396. <VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
144	**397. <VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Exclusão Venda
145	288. <VEND, (13106, <62, 100>), {VEND, CLI, FUNC}, (11119, (75, 83))>
146	**289. <VEND, (13106, <62, 100>), {VEND, CLI, FUNC}, (11119, (75, 232))>**
147	290. <VEND, (13106, <62, 100>), VEND, (11116, (70, 77))>
148	**291. <VEND, (13106, <62, 100>), VEND, (11116, (70, 156))>**
149	<VEND, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 77))>
150	**<VEND, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 329))>**
151	294. <VEND, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (11119, (75, 83))>
152	**295. <VEND, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (11119, (75, 232))>**
153	400. <VEND, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
154	**401. <VEND, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Inclusão Cliente/Inclusão Venda
155	304. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (11119, (75, 83))>
156	**305. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC}, (11119, (75, 232))>**
157	308. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), VEND, (11116, (70, 77))>
158	**309. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), VEND, (11116, (70, 156))>**
159	306. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 77))>
160	**307. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 329))>**
161	312. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (11119, (75, 83))>
162	**313. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (11119, (75, 232))>**
163	420. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
164	**421. <CLI, (13104, <62, 78>), VEND, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	TABELA ITENS (Dependência com Tabela Venda e Produto)
	Inclusão Vend/Itens
165	318. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 77))>
166	**319. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 329))>**
167	320. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {ITEM, PROD}, (11112, (74, 81))>
168	**321. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {ITEM, PROD}, (11112, (74, 228))>**
169	322. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), ITEM, (11117, (70, 77))>
170	**323. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), ITEM, (11117, (70, 156)>**
171	404. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
172	**405. <VEND, (13104, <62, 78>), <ITEM, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Exclusão Itens
173	328. <ITEM, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 77))>
174	**329. <ITEM, (13106, <62, 100>), {VEND, ITEM, FUNC, CLI}, (11110, (70, 329))>**

175	330. <ITEM, (13106, <62, 100>), {ITEM, PROD}, (11112, (74, 81))>
176	**331. <ITEM, (13106, <62, 100>), {ITEM, PROD}, (11112, (74, 228))>**
177	332. <ITEM, (13106, <62, 100>), ITEM, (11117, (70, 77))>
178	**333. <ITEM, (13106, <62, 100>), ITEM, (11117, (70, 156))>**
179	408. <ITEM, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
180	**409. <ITEM, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	TABELA PAGAMENTO (Dependência com Tabela Venda)
	OBS: a tabela pagamento é definida juntamente com tabela venda
	Inclusão Vend/Pagamento
181	342. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (11119, (75, 83))>
182	**343. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG}, (11119, (75, 232))>**
183	344. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (11111, (70, 77))>
184	**345. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (11111, (70, 167))>**
185	346. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (11118, (70, 77))>
186	**347. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (11118, (70, 156))>**
187	348. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (11110, (221, 230))>
188	**349. <VEND, (13104, <62, 78>), PAG, (13104, <62, 78>), PAG, (11110, (221, 329))>**
189	412. <PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
190	**413. <PAG, (13104, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**
	Exclusão Pagamento
191	360. <PAG, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (11119, (75, 83))>
192	**361. <PAG, (13106, <62, 100>), {VEND, CLI, FUNC, PAG}, (11119, (75, 232))>**
193	362. <PAG, (13106, <62, 100>), PAG, (11111, (70, 77))>
194	**363. <PAG, (13106, <62, 100>), PAG, (11111, (70, 167))>**
195	364. <PAG, (13106, <62, 100>), PAG, (11110, (221, 230))>
196	**365. <PAG, (13106, <62, 100>), PAG, (11110, (221, 329))>**
197	366. <PAG, (13106, <62, 100>), PAG, (11118, (70, 77))>
198	**367. <PAG, (13106, <62, 100>), PAG, (11118, (70, 156))>**
199	416. <PAG, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 83))>
200	**417. <PAG, (13106, <62, 78>), {VEND, CLI, FUNC, PAG, ITEM}, (11119, (75, 232))>**

ANEXO II - Grafos das classes Banco.Java e Operacoes.java



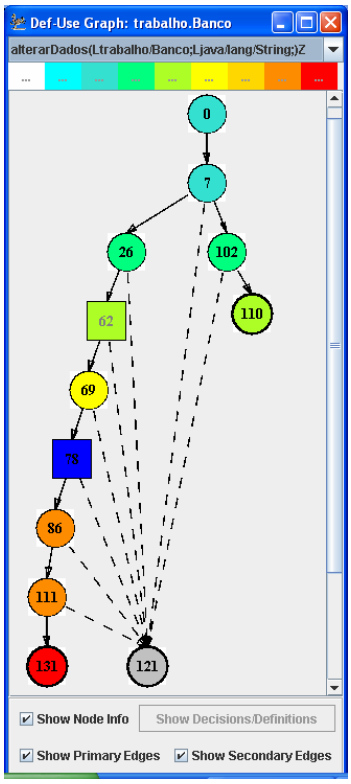
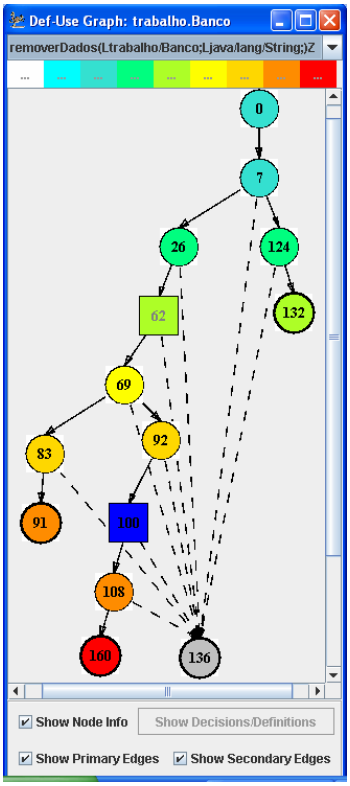


Figura 1 e 2: Grafos referentes aos Métodos inserirDados (13104) e alteraDados (13105), ambos da classe Banco.java



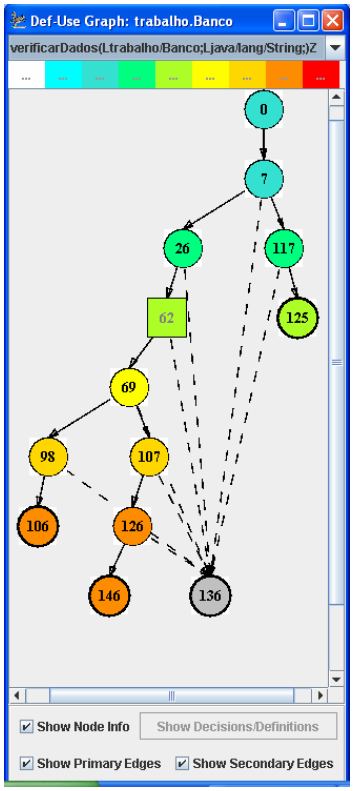
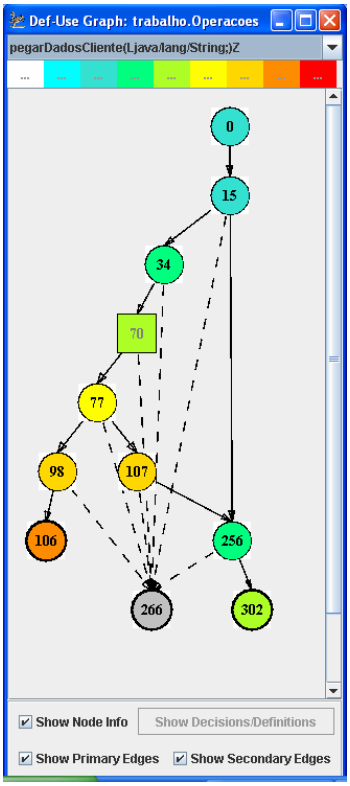


Figura 3 e 4: Grafos referentes aos Métodos removerDados (13106) e verificarDados(13108), ambos da classe Banco.java



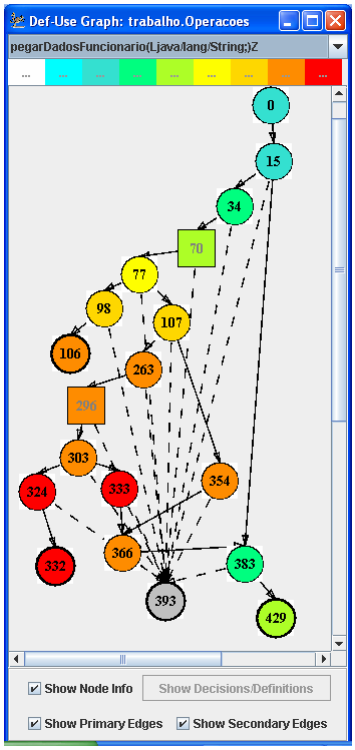
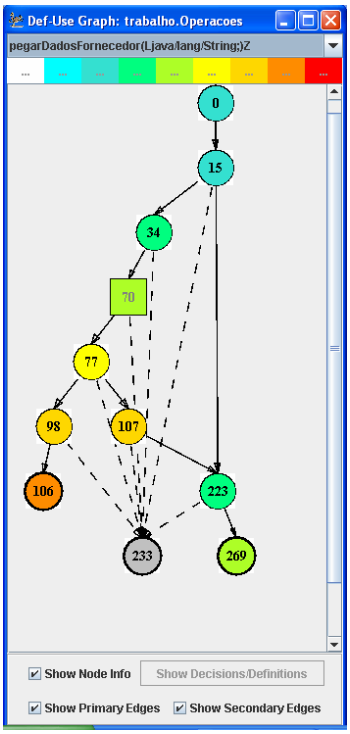


Figura 5 e 6: Grafos referentes aos Métodos pegarDadosCliente (11105) e pegarDadosFuncionario(11106), ambos da classe Operacoes.java



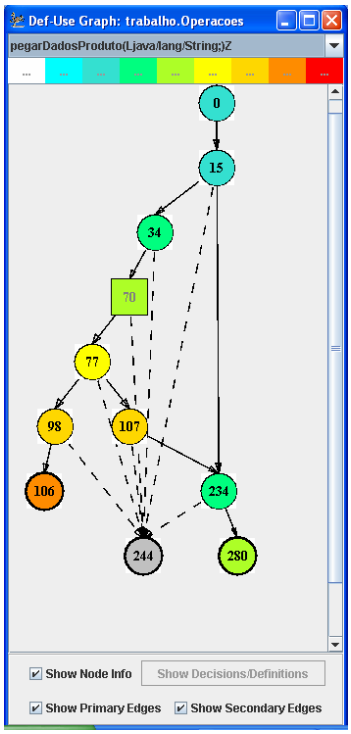
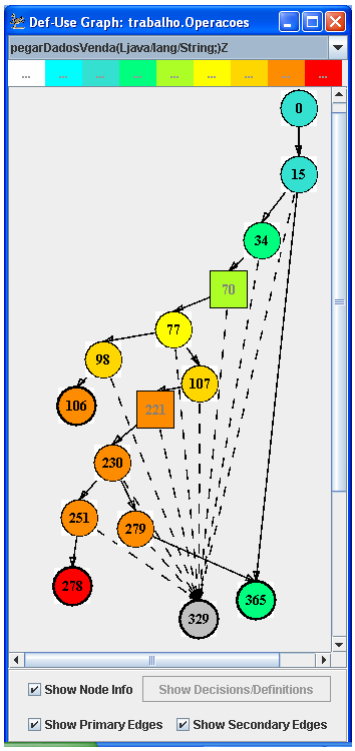


Figura 7 e 8: Grafos referentes aos Métodos pegarDadosFornecedor (11107) e pegarDadosProduto(11108), ambos da classe Operacoes.java



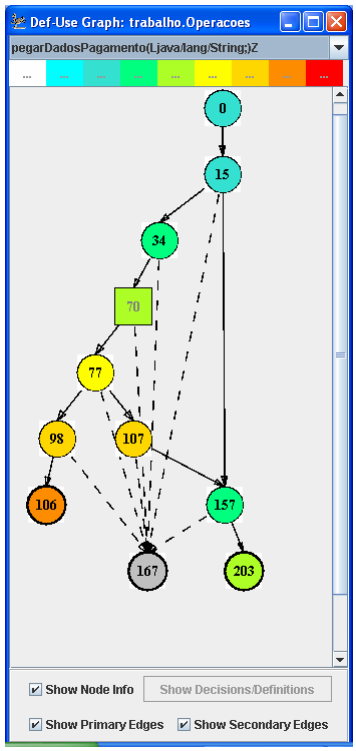
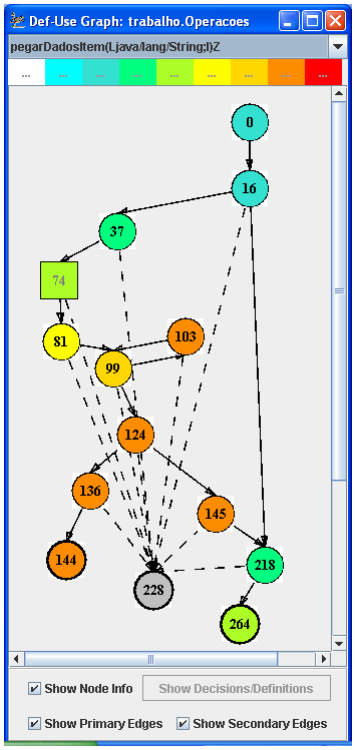


Figura 9 e 10: Grafos referentes aos Métodos pegarDadosVenda (11110) e pegarDadosPagamento (11111), ambos da classe Operacoes.java



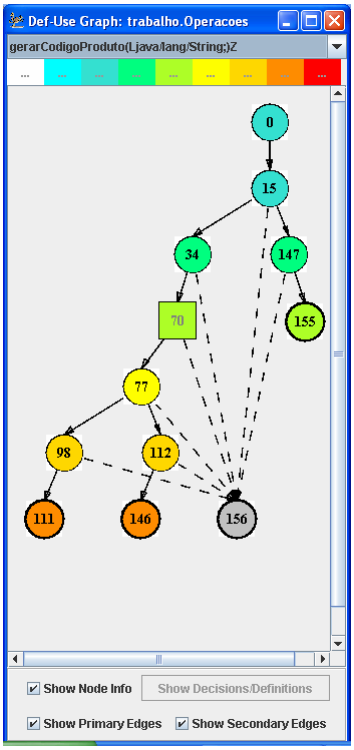
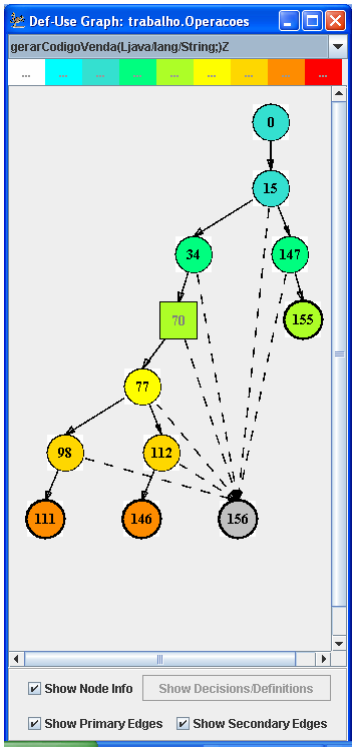


Figura 11 e 12: Grafos referentes aos Métodos pegarDadosItem (11112) e gerarCodigoProduto (11114), ambos da classe Operacoes.java



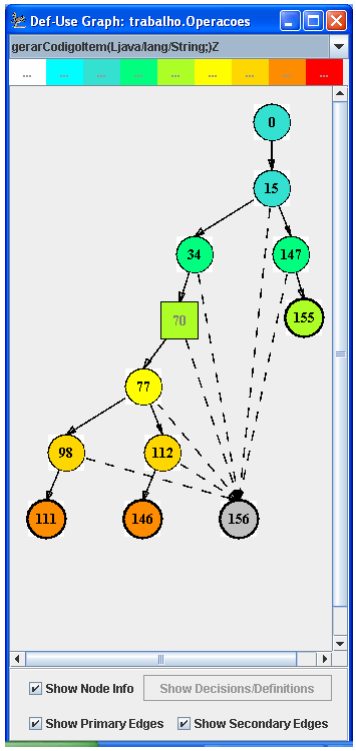
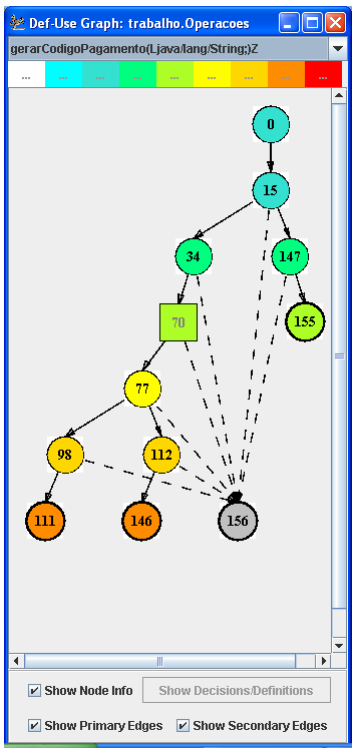


Figura 13 e 14: Grafos referentes aos Métodos gerarCodigoVenda (11116) e gerarCodigoItem (11117), ambos da classe Operacoes.java



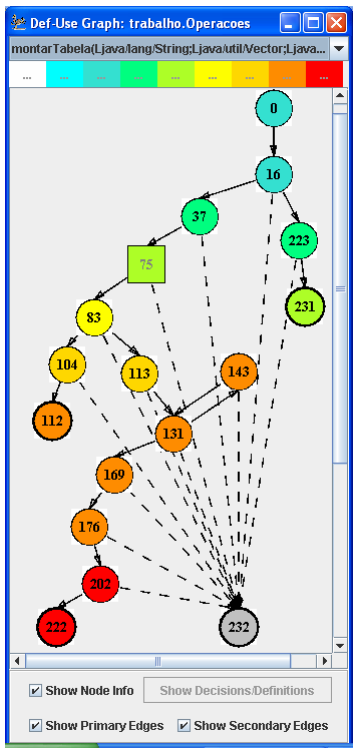


Figura 15 e 16: Grafos referentes aos Métodos gerarCodigoPagamento (11118) e montarTabela (11119), ambos da classe Operacoes.java