"FUNDAÇÃO DE ENSINO EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA" - UNIVEM

RENATO GERALDI

CRIAÇÃO DE UM *WORKFLOW* DE TESTE VOLTADO PARA BANCO DE DADOS

MARÍLIA
2006

RENATO GERALDI

CRIAÇÃO DE UM *WORKFLOW* DE TESTE VOLTADO PARA BANCO DE DADOS

Monografia de Conclusão de Curso, apresentada ao Centro Universitário Eurípides de Marília – UNIVEM – para a obtenção do Título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Edmundo Sérgio Spoto.
Co-orientadora: Prof.ª Dr.ª Maria Istela Cagnin Machado.

Área de Concentração

Engenharia de *Software*

UNIVEM – MARÍLIA
2006

Ao Deus todo poderoso que me deu toda força e inspiração necessária para o desenvolvimento deste trabalho e aos meus familiares e amigos que me apoiaram.

AGRADECIMENTOS

Agradeço primeiramente a Deus que me deu o Dom da Vida, Saúde, Inteligência, Sabedoria e tudo mais que eu necessitei.

Aos meus familiares, de forma especial a minha irmã Daniele Geraldi, a minha mãe Sônia Fátima Gonçalves, a minha avó Ana Perinetti Gonçalves e a minha noiva Leiliane Gabriel de Souza, que muito me apoiaram em todos os sentidos.

Aos amigos Jeferson Davi Parckert e José Ricardo Sarmento pela paciência e ajuda.

A todos os professores da instituição que me auxiliaram de forma direta ou indireta, dirimindo minhas dúvidas, contribuindo assim para o desenvolvimento deste trabalho.

A Prof.ª Dr.ª Fátima L. Santos Nunes Marques, coordenadora do curso de Ciência da Computação e ao Prof. Dr. José Remo Ferreira Brega, coordenador do programa de TCC do curso.

Especialmente ao Prof. Dr. Edmundo Sérgio Spoto meu orientador, e a Prof.ª Dr.ª Maria Istela Cagnin minha co-orientadora, que tiveram muita atenção e paciência comigo e foram fundamentais para a realização deste trabalho.

GERALDI, Renato. **Criação de um *Workflow* de Teste voltado para Banco de Dados**. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília - UNIVEM, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

RESUMO

A necessidade de aperfeiçoamento de técnicas e ferramentas, essenciais para a execução de teste de *Software* em Aplicações de Banco de Dados Relacional (ABDR) tem evoluído, mas diferentemente das técnicas e ferramentas voltadas para o desenvolvimento de *Software*, de tal forma que, estas têm evoluído mais rapidamente. Como parte dessa evolução existem conceitos desenvolvidos recentemente, como o critério todos *t-usos* proposto por Spoto (2000), que se refere à técnica de teste estrutural baseado no modelo fluxo de dados voltado para ABDRs, tratando especificamente as ocorrências de comandos SQL no código fonte de um sistema. Este trabalho apresenta a elaboração de um *Workflow* contendo todos os passos necessários à aplicação de teste de *Software* em ABDRs, baseado na Norma IEEE-std-829-1998 (que basicamente regulamenta a documentação de teste de *Software*) e no critério supramencionado. A proposta deste trabalho vai ao encontro dessa necessidade, tornando a execução de teste de *Software* em ABDRs mais eficiente e menos custosa.

Palavras-chave: Teste de *Software*. Teste Estrutural. *Workflow* de Teste. ABDR. Todos *t-usos*. Norma IEEE 829.

GERALDI, Renato. **Criação de um *Workflow* de Teste voltado para Banco de Dados**. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília - UNIVEM, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

ABSTRACT

The necessity of improvement of techniques and tools, essentials for the execution of a Software test in Relational Databases Application (RDA) has been suffering evolution, but distinct from the techniques and tools used for the Software development, in a way that, these have evoluted fastly. As part of this evolution there are concepts which were developed recently, like the criterium all the *t-uses* proposed by Spoto (2000), which refers to the technique of structural test based on the data flow model used for RDAs, treating specifically the occurrences of commands SQL in the source code of a system. This work presents the accomplishment of a Workflow containing all the steps needed to the application of a Software test in RDAs, based on the rule IEEE-std-829-1998 (which basically guides the rules for the documentation of this Software test) and in the criterium mentioned before. The proposal of this work intends to supply this necessity, making the execution of this software test in RDAs much more efficient and less expensive to be accomplished.

Keywords: Software Test. Structural Test. Teste Workflow. RDA. All t-uses. Rule IEEE 829.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABDR: Aplicação de Banco de Dados Relacional.

BD: *Data Base* (Banco de Dados).

CBD: Component Based Development (Desenvolvimento Baseado em Componentes).

DML: *Data Manipulation Language* (Linguagem de Manipulação de Dados).

FADAT: Ferramenta de Apoio à Documentação da Aplicação de Teste de *Software*

IEEE: *Institute of Electrical and Electronics Engineers* (Instituto de Engenheiros Eletricistas e Eletrônicos).

JaBUTi: *Java Bytecode Understanding and Testing.*

OO: Orientado a Objeto.

RUP: *Rational Unified Process* (Processo Unificado da Rational).

SGBD: Sistema Gerenciador de Banco de Dados.

SQL: *Structured Query Language* (Linguagem de Consulta Estruturada).

UP: Unidade de Programa.

V&V: Verificação e Validação.

SUMÁRIO

INTRODUÇÃO

Com a evolução das técnicas e ferramentas de desenvolvimento de *Software*, as técnicas e ferramentas voltadas para o desenvolvimento de teste de *Software* têm a necessidade de evoluírem também, fato este que não tem acontecido eficientemente.

O teste de *Software* é um elemento crítico da garantia de qualidade de *Software* e representa a revisão final da especificação, projeto e geração de código. A crescente visibilidade do *Software* como um elemento do sistema e os "custos" de atendimento associados com uma falha, são forças motivadoras para o teste rigoroso e bem planejado (PRESSMAN, 2005).

As atividades de teste consomem em média 50% do tempo e do custo de desenvolvimento de um produto de *Software*. Esse gasto é em parte resultante da escassez de ferramentas de auxílio ao teste que permitam uma forma planejada e segura para sua execução e para a avaliação de seus resultados (SPOTO, 2000).

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).

Segundo Spoto, *et al.* (2005), o teste de *Software* envolve de forma geral, as seguintes atividades: planejamento, projeto de casos de teste, execução e avaliação dos resultados dos testes. Os métodos utilizados para testar *Software* são baseados essencialmente nas técnicas funcional, estrutural e baseada em defeitos.

A técnica funcional de teste visa a estabelecer requisitos de teste derivados da especificação funcional do *Software* (teste caixa preta) enquanto a técnica estrutural de teste apóia-se em informações derivadas diretamente da implementação (teste caixa branca) (WHITE, 1987).

Um caso de teste é o conjunto formado por: um dado de entrada para um determinado programa e uma saída esperada para esse dado de entrada. Em geral, os casos de teste são projetados a fim de se obter a maior probabilidade de encontrar a maioria dos defeitos com um mínimo de esforço e tempo (SPOTO, 2000).

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).

Segundo Chays *et al.* (2000), 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 trate o teste em programas de aplicação de banco de dados, motivando assim este trabalho.

O objetivo deste trabalho é estudar e analisar as etapas de testes de *Software* de uma ABDR, visando apoiar a construção de um *Workflow* de teste para o desenvolvimento de *Software* e conduzir um roteiro das etapas de testes exigidas pela Norma IEEE 829 de 1998, baseado no modelo de teste de *Software* do RUP.

O teste de um *Software* é uma das formas de obtenção da qualidade de *Software*, sendo assim, este trabalho auxiliará as equipes de teste de *Software*, com intuito de melhorar e planejar as tarefas de testes que envolvem o nível de unidade.

O *Workflow* de teste, fruto dos resultados deste trabalho, servirá como uma ferramenta de orientação e apoio ao teste de *Software* em sistemas que utilizam BD e SGBD, independentemente do paradigma ser OO ou Relacional.

É importante observar que este trabalho concentra-se em teste de *Software* baseado na técnica estrutural com enfoque em fluxo de dados em ABDRs.

Visto a importância do teste na qualidade de um *Software,* e a falta de atenção que a equipe de desenvolvimento tem por esta fase de teste – que é essencial no desenvolvimento de um *Software* com qualidade – é que adquiri motivação para a execução deste trabalho.

Esta desatenção se dá, na maioria dos casos, em função da fase de teste ser custosa e trabalhosa. A carência de ferramentas de tecnologia na área de desenvolvimento de ferramentas de teste de *Software* é também um fator agravante para a não importância à fase de teste de um *Software*.

Segue a estrutura organizacional deste trabalho, de tal forma que:

Na Introdução são descritas as principais visões introdutórias do assunto como a Introdução propriamente dita, o Objetivo, a Motivação e a Organização da Monografia.

No Capítulo 1 são apresentados os Conceitos e Terminologias de Testes de *Software*, que envolvem algumas Técnicas como: Teste Funcional, Teste Estrutural e Testes Baseados em Erros. Nesse Capítulo são apresentados também alguns conceitos referentes à aplicação do Teste Estrutural em ABDRs.

No Capítulo 2 são apresentados os procedimentos para o desenvolvimento de uma etapa de Teste de *Software* bem como da apresentação de Normas que controlam as etapas de Teste e sobre a geração de um *Workflow* de Teste.

No Capítulo 3 são apresentados conceitos genéricos de *Workflow* e alguns modelos de *Workflow* voltado para o Teste de *Software* como: *Workflow* de Teste do RUP e *Workflow* de Teste da Ferramenta FADAT.

No Capítulo 4 são apresentados os passos e procedimentos envolvidos na forma de um *Workflow* de Teste, sendo este o propósito do trabalho. Nesse Capítulo são apresentadas também, algumas telas que ilustram a inserção das informações do *Workflow* proposto neste trabalho na Ferramenta FADAT. Paralelamente ao *Workflow* de Teste, são apresentados: *i)* informações utilizadas em fase experimental, que contribuíram para o desenvolvimento deste

trabalho e, *ii)* Artefatos gerados nos passos deste *Workflow,* com base na Norma IEEE-std-829-1998.

Ao final são apresentadas as Conclusões e informações referentes a Trabalhos Futuros, que poderão ser desenvolvidas a partir deste trabalho; as Referências Bibliográficas, fonte de dados e informações importantes para o desenvolvimento deste Trabalho, e os APÊNDICES A e B, que contém respectivamente o Código de Programa referente ao Estudo de Caso deste Trabalho e o Código de Programa referente ao Estudo de Caso Instrumentado através da Ferramenta de Teste JaBUTi

1 – REVISÃO BIBLIOGRÁFICA

1.1 – Conceitos de Teste de *Software*

O teste de um *Software* consiste em verificar se todos os processos executados até o momento estão corretos, esta fase utiliza-se de várias técnicas – que serão mencionadas no decorrer deste Capítulo – com intuito de descobrir erros no sistema ainda não descobertos.

O processo de teste consiste basicamente em introduzir dados de entrada necessários à execução do programa e, após sua execução, comparar os resultados obtidos com os resultados esperados (SPOTO, *et al.*, 1995).

O objetivo essencial do teste é o de revelar os defeitos existentes no programa. Conforme define Myers (1979) "Teste é o processo de executar o programa com a intenção de detectar erros".

É interessante neste momento, explicitar alguns conceitos utilizados no trabalho (SPOTO, *et al.*, 1995):

- Uma falha é um evento notável – ou perceptível – em que o sistema viola a sua especificação.
- Um defeito no programa é uma deficiência algorítmica que pode levar a uma falha do sistema.
- Um erro é um estado incorreto do conjunto de dados do programa, que pode ser manifestado, ocasionando assim uma falha.

Na literatura, muitas vezes, utiliza-se a palavra erro neste sentido, mas também, em alguns casos, como sinônimo de defeito.

A fase de teste de um *Software* é a que mais exige esforços de todo o processo de desenvolvimento, é justamente por este motivo que já no desenvolvimento do sistema deve-se enfatizar a testabilidade do mesmo, facilitando ao máximo as formas de testar o sistema.

Esta fase exige um custo alto, por este motivo, grande parte das organizações não atenta para o teste, que é um ponto fundamental na garantia de qualidade do *Software*. Os testes de *Software* de tempo real que na sua grande maioria envolvem vidas, podem custar mais caro ainda, em função de sua aplicação (SOMMERVILLE, 2003).

Um teste para ser mais eficiente e eficaz precisa ser planejado com antecedência, pode ter seu início na fase de análise e levantamento de requisitos, antes mesmo de iniciar a fase de projeto.

Para termos a certeza de que um *Software* está adequado ao seu propósito, ou seja, cumpra com suas especificações e atenda às necessidades do cliente que o solicitou, deve-se utilizar os processos de V&V, que consiste num ciclo de vida bem definido, iniciando com as revisões dos requisitos, passando pelas revisões de projeto e inspeções de código até chegar aos testes do sistema (SOMMERVILLE, 2003).

Embora haja semelhança no significado, verificação e validação não são a mesma coisa, pode-se entender isso através da explicação simplificada, expressa por BOEHM (1979):

- Validação: Estamos Construindo o produto certo?
- Verificação: Estamos Construindo certo o produto?

O processo de V&V limita-se única e exclusivamente em estabelecer a existência de defeitos em um processo de *Software*, enquanto que a depuração limita-se em localizar e corrigir defeitos. Depois de depurado o sistema deve ser reavaliado, ou seja, reinspecionado ou retestado, estes são chamados de testes de regressão servem para checar se as mudanças feitas no programa estão corretas ou se não introduziram novos erros (SOMMERVILLE, 2003).

Os testes podem ser conduzidos em três níveis (PRESSMAN, 2000):

Teste de Unidade: concentra esforços na menor unidade do projeto de *Software* (módulo), ou seja, procura identificar erros de lógica e de implementação em cada módulo do *Software* separadamente;

Teste de Integração: é uma técnica sistemática para a construção de estrutura de programa, realizando-se, ao mesmo tempo, teste para descobrir erros associados às interfaces. O objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa, que foi determinada pelo projeto;

Teste de Sistema: é na verdade uma série de diferentes testes cujo objetivo é verificar se todos os elementos do sistema foram adequadamente integrados e realizam as funções atribuídas.

As técnicas de teste funcional e teste estrutural podem ser aplicadas para um módulo (teste de unidade), para a integração dos módulos (teste de integração), para a validação dos requisitos (teste de validação), para o sistema como um todo (teste de sistema) e para a fase de manutenção (teste de regressão). É importante lembrar que nesse contexto, a unidade refere-se ao código de um pequeno componente do *Software*, a fase de integração considera as unidades juntas, o sistema é o contexto final do qual o *Software* faz parte e o teste de regressão aplica-se após modificações efetuadas no *Software* e também em novas versões do produto (CRESPO, *et al*., 2000).

Em geral, não se consegue por meio de testes provar que um programa está correto, portanto, testar contribui no sentido de aumentar a confiança de que o *Software* executa corretamente as funções especificadas. Apesar das limitações próprias da atividade de teste, sua aplicação de maneira sistemática e bem planejada pode garantir ao *Software* algumas características mínimas que são importantes tanto para o estabelecimento da qualidade do produto quanto para seu processo de evolução (DELAMARO, 1993).

1.1.1 – Teste Funcional

A técnica de teste funcional chamada ainda de teste caixa preta ou teste comportamental, aborda os testes como derivações da especificação de programa ou componente. O sistema funciona como se fosse uma caixa preta, onde são estudadas somente as entradas e saídas, sendo o foco principal a funcionalidade do sistema e não a implementação do *Software*.

O teste funcional testa o sistema do ponto de vista do usuário, isto é, não considera a estrutura interna ou a forma de implementação do sistema, olhando-se o *Software* apenas através de suas interfaces. Sendo este o único tipo de teste possível quando não se dispõem do código-fonte do sistema. Os erros encontrados através do teste funcional são: erros de interfaces, funções incorretas, erros na estrutura de dados ou no acesso a dados externos, erros de desempenho e erros de iniciação ou finalização (CRESPO, *et al*., 2000).

Dadas às entradas, as saídas são examinadas, se as saídas não são aquelas previstas, pode-se dizer que o teste detectou com sucesso um problema no *Software*. O problema enfrentado pelos responsáveis dos testes é selecionar as entradas que tenham grande possibilidade de provocarem comportamento anômalo.

Além de demonstrar a operacionalidade das funcionalidades do sistema, e adequação da saída em relação à entrada o teste caixa preta serve também para demonstrar que a integridade da informação externa, por ex.: uma base de dados é mantida (PRESSMAN, 2005).

Segundo Crespo, *et al*. (2000), o teste funcional compreende critérios de teste voltados para a funcionalidade do *Software*. Os principais critérios para a seleção dos dados de testes associados à técnica funcional são:

- **Particionamento de Equivalência:** direciona o testador a construir dados de testes oriundos das classes de dados do domínio de entrada do programa.
- **Análise de Valor Limite:** direciona o testador a construir dados de teste oriundos das "extremidades" do domínio de entrada do programa, ou seja, valores fronteiriços.
- **Grafo de Causa-Efeito:** direciona o testador a construir dados de teste de maneira que todas as *condições lógicas* de entrada do programa, representadas num grafo de *causa e efeito,* sejam testadas.

1.1.2 – Teste Estrutural

O teste estrutural ou teste caixa branca consiste numa abordagem de testes que são derivados do conhecimento da estrutura e da implementação do *Software*, para diferenciá-los do teste caixa preta, podem ainda ser chamados também de testes de caixa de vidro ou testes de caixa clara (SOMMERVILLE, 2003).

Os testes de caixa branca são geralmente aplicados às unidades de programas relativamente pequenas como sub-rotinas, ou às operações associadas com um objeto.

Nesta abordagem o testador pode analisar o código, consequentemente conhecendo a estrutura poderá derivar os dados para o teste e determinar quantos casos de teste serão necessários para que todas as declarações no programa ou componente sejam executadas pelo menos uma vez durante o teste.

O teste estrutural é realizado com a finalidade de verificar as estruturas internas do *Software*. Quando se conhece a estrutura interna do *Software*, o teste pode ser conduzido para verificar se todos os componentes internos, quando exercitados, operam de maneira adequada. Através deste teste pode-se garantir que todos os passos de um dado módulo foram

executados pelo menos uma vez, podem-se exercitar todas as decisões lógicas, ou executar todos os comandos iterativos nos seus valores limites e em sua faixa operacional (CRESPO, *et al*., 2000).

Utilizando a abordagem caixa branca, o engenheiro de *Software* pode derivar casos de teste que satisfaz os seguintes critérios (PRESSMAN, 2005):

1 – Casos de teste que garantam que todos os caminhos independentes de um módulo tenham sido executados pelo menos uma vez,

2 – Casos de teste que exercitam todas as decisões lógicas em seus lados verdadeiro e falso,

3 – Casos de teste que executam todos os ciclos nos seus limites e dentro de seus intervalos operacionais e,

4 – Caso de testes que exercitam as estruturas de dados internas para garantir sua validade.

É importante salientar que não se pode pensar que é perda de tempo aplicar os testes caixa branca, por serem mais minuciosos, em vez de aplicarmos somente os testes caixa preta, pois os testes caixa branca devem ser aplicados pelos seguintes motivos (PRESSMAN, 2005):

1 – Erros Lógicos e pressupostos incorretos são inversamente proporcionais à probabilidade de que um caminho de programa vai ser executado.

2 – Frequentemente acredita-se que um caminho lógico não é provável de ser executado quando, na realidade, ele pode ser executado em base regular.

3 – Erros Tipográficos são aleatórios. Quando um programa é traduzido em código fonte, numa linguagem de programação, é provável que ocorram alguns erros de digitação. É provável que um erro tipográfico exista tanto num caminho lógico obscuro quanto num caminho principal.

Inicialmente pode-se representar a estrutura interna de um sistema através de um grafo dirigido, com somente um nó de entrada e saída. Cada um desses nós é considerado como uma seqüência de comandos que são executados como um bloco de comandos, cada arco representa uma transferência de controle entre esses blocos e um caminho de programa pode ser representado como uma seqüência de nós (FRANKL, 1998; WEYUKER, 1998).

Um bloco é uma seqüência máxima de comandos, no qual, se o primeiro comando for executado, necessariamente os demais comandos subseqüentes no bloco.

O teste estrutural, de acordo com seu enfoque, pode ser dividido em duas partes (SPOTO, *et al.*, 1995):

1 – Técnica Estrutural baseada em Fluxo de Controle: nesta técnica a base para a seleção de dados de teste é feita através de informações de fluxo de controle do programa, de tal forma que determinados tipos de estrutura do grafo do programa sejam exercitadas.

2 – Técnica Estrutural baseada em Fluxo de Dados: nesta técnica, são utilizados informações do fluxo de dados existentes no programa, com intuito de identificar atribuições de utilizações das variáveis através do programa, dessa forma gerando componentes elementares a serem exercitados pelo testador.

Segundo Crespo, *et al*. (2000), o teste estrutural compreende critérios voltados para a estrutura de implementação do *Software*, sendo que nesta técnica os critérios também são divididos em 2 grupos, que são apresentados a seguir:

1) Testes baseados em Fluxo de Controle – compreendem critérios de testes fundamentados nas estruturas de controle do código:
    - **Teste de Comandos:** direciona o testador a construir dados de teste de maneira que cada comando do código seja executado pelo menos uma vez.
    - **Teste de Ramos:** direciona o testador a construir dados de teste de maneira que cada ramo de decisão no programa seja testado pelo menos uma vez.

- **Teste de Condições:** direciona o testador a construir dados de teste de tal maneira que cada condição dentro de uma decisão seja testada pelo menos uma vez.
- **Teste de Condições Múltiplas:** direciona o testador a construir dados de teste de tal maneira que cada combinação das condições dentro de uma decisão seja testada pelo menos uma vez.
- **Teste de Laços:** direciona o testador a construir dados de teste de tal maneira que todos os laços do programa sejam testados pelo menos uma vez.

2) Testes baseados em Fluxo de Dados – compreendem critérios de testes fundamentados nos tipos de ocorrência de uma variável, basicamente, em sua *definição* e seu *uso*:

- **Todos C-Usos:** direciona o testador a construir dados de teste de tal maneira que cada *associação definição c-uso* de uma variável no programa seja exercitada pelo menos uma vez, onde c-*uso* representa o uso de uma variável numa computação (num cálculo).
- **Todos P-Usos:** direciona o testador a construir dados de teste de tal maneira que cada *associação definição p-uso* de uma variável no programa seja exercitada pelo menos uma vez, onde *p-uso* representa o uso de uma variável num predicado (comandos do tipo *If, While, Until e Case*).
- **Todos Usos:** direciona o testador a construir dados de teste de tal maneira que cada *associação definição-uso* de uma variável no programa seja exercitada pelo menos uma vez, onde *uso* representa o uso de uma variável numa computação ou num predicado.

Um problema relacionado ao teste estrutural é a impossibilidade de determinar se um caminho é ou não executável, ou seja, não existe algoritmo que dado um caminho completo

qualquer, forneça o conjunto de valores que causam a execução desse caminho. Desta forma, é necessária a intervenção do testador para determinar quais são os caminhos não executáveis para o programa que está sendo testado (DORIA, 2001).

1.1.3 – Teste Baseado em Erros

A técnica de teste baseada em erros, utiliza informações sobre os tipos de erros mais freqüentes no processo de desenvolvimento de *Software*, para derivar os requisitos de *Software*. A ênfase da técnica está nos erros que o programador pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar sua ocorrência. Os critérios semeadura de erros (*Error Seeding*) e análise de mutantes (*Mutation Analysis*) são típicos de técnica baseada em erros (MALDONADO, *et al.*, 1998).

A análise de mutantes utiliza um conjunto de programas ligeiramente modificados (mutantes), obtidos a partir de um determinado programa P, com intuito de avaliar o quanto um conjunto de casos de teste T é adequado para o teste desse programa P. O objetivo é encontrar um conjunto de casos de teste que consiga revelar, por meio da execução de P, as diferenças de comportamento existentes entre P e seus mutantes (MALDONADO, *et al.*, 1998).

Foi publicado em 1978 por Demillo (1978), um dos primeiros artigos que descrevem as idéias de testes de mutantes. A idéia básica da técnica de Demillo, conhecida como a hipótese do programador competente, assume que programadores experientes escrevem programas muito próximos do correto. Com base na validade dessa hipótese, pode-se afirmar que, erros são introduzidos no programa por meio de pequenos desvios sintáticos que, embora não causem erros sintáticos, alteram a semântica do programa, e consequentemente, conduzem o programa a um comportamento incorreto.

Para revelar tais erros, a análise de mutantes identifica os desvios sintáticos mais comuns, e através de pequenas transformações sobre o programa em teste, encoraja o testador a construir casos de teste que mostrem que tais transformações levam a um programa incorreto (MALDONADO, *et al.*, 1998).

Uma outra hipótese utilizada na aplicação do critério análise de mutantes é o efeito de acoplamento (*Coupling Effect*), na qual assume que erros complexos estão relacionados a erros simples. Assim sendo, espera-se, e alguns estudos empíricos já confirmaram essa hipótese (ACREE, *et al.*, 1979; BUDD, *et al.*, 1980), que conjuntos de casos de teste capazes de revelar erros simples são também capazes de revelar erros complexos. Nesse sentido, aplica-se uma mutação de cada vez no programa P em teste, ou seja, cada mutante contém apenas uma transformação sintática (MALDONADO, *et al.*, 1998).

Baseado na hipótese do programador competente e do efeito de acoplamento pode-se descrever a analise de mutantes da segunda forma:

A princípio, o testador deve fornecer um programa P a ser testado e um conjunto de casos de teste T cuja adequação se deseja avaliar. O programa é executado sobre T e, se apresentar resultados incorretos, então um erro foi encontrado e o teste termina. Caso contrário, o programa ainda pode conter erros que o conjunto T não conseguiu revelar. O programa P sofre então pequenas alterações, dando origem aos programas P1, P2... Pn, que são mutantes de P, diferindo deste, apenas ocorrência de erros simples, ou seja, cada mutante contém apenas uma mutação (DORIA, 2001).

1.2 – Teste Estrutural de Aplicação de Banco de Dados Relacional (ABDR)

Na execução dos testes em um programa de ABDR, é necessário atentar-se para o estado do BD antes e depois de cada execução de um caso de teste (CHAYS, *et al.*, 2000).

Isso se faz necessário, porque uma falha causada por um caso de teste anterior, pode modificar o estado da relação, de forma a tornar o BD inconsistente, influenciando o resultado do caso de teste atual. As alterações que ocorrem no BD, à qualidade da informação armazenada e sua confiabilidade são itens imprescindíveis para análise do BD.

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 (BATISTA, 2003).

Há características semelhantes entre teste de programas convencionais e teste estrutural de programas de ABDR, adaptando nesta, comandos SQL, variáveis de tabela e *variáveis host* (quando estas diferem das variáveis de programa). No caso da linguagem Java pode-se considerar a *variável host* sendo a mesma variável de programa (BECARI, 2005).

Segundo Spoto (2000), existem 3 tipos de variáveis utilizadas numa ABDR: *variáveis de programa* (variáveis da linguagem hospedeira) $\boldsymbol{P = \{p_1, p_2, .., p_m\}}$; *variáveis de ligação* ou *variáveis host* (que guardam os valores da base de dados) $\boldsymbol{H = \{h_1, h_2, .., h_n\}}$; e as *variáveis tabela,* $\boldsymbol{T = \{t_1, t_2, .., t_k\}}$, que são as tabelas da base de dados relacional manipuladas pelo programa.

As *variáveis tabela de visão* são extraídas das *variáveis tabela* da base de dados da aplicação. A ocorrência de uma *variável host* ou de uma *variável de programa* pode ser: uma *definição* de variável, um *uso* de variável ou uma *indefinição*. As *variáveis tabela* são variáveis globais para todos os Módulos de Programas Mod_i de uma ABDR.

As *variáveis tabela* são criadas através do comando CREATE TABLE <*tabela*>, e essa criação é feita uma única vez durante o projeto do BD. A partir de sua criação, o SGBD passa a controlar os acessos às tabelas da ABDR, permitindo que seus usuários possam modificá-las e/ou atualizá-las a partir dos programas da aplicação. Assim, as *variáveis tabela*

não são declaradas como as *variáveis host* e *variáveis de programa* nos programas de aplicação (SPOTO, 2000).

Desse modo, adota-se a ocorrência de uma *variável tabela* ***t*** como sendo *definição* ou *uso* e considera-se que toda *variável tabela* referenciada por algum programa implica a ocorrência de uma definição anterior (até por 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*" (ELMASRI, 1994; NAVATHE, 1994).

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).

Os comandos executáveis da Linguagem SQL (INSERT, DELETE, UPDATE, SELECT, COMMIT, ROLLBACK, entre outros) são acomodados isoladamente. Foram adotados como notação gráfica, nós circulares para representar os blocos de comandos e nós retangulares para representar os comandos executáveis da SQL (NARDI, *et al.*, 2005), podendo ser visualizados no grafo representado na Figura 1.1 (BECARI, 2005). Nesta Figura o método *InserirDados()* possui os nós retangulares 62 e 78, que representam os comandos da SQL INSERT e COMMIT, respectivamente.

As setas, denominadas de arcos, representam possíveis transferências de controle entre os nós (NARDI, 2006).

De forma geral, nos testes de programas convencionais, a escolha dos dados de testes, baseado nos elementos requeridos dos critérios de fluxo de dados, 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 (BECARI, 2005).

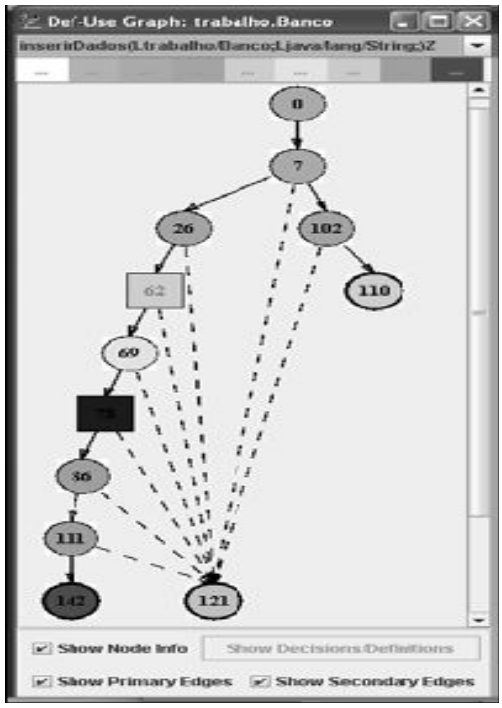

Figura 1.1 – Grafo do Método *InserirDados()*.

De acordo com Spoto (2000), o principal objetivo do teste estrutural de programas de aplicação é detectar a presença de defeitos relacionados à implementação, porém, existem situações em que o teste em ABDR exercita:

- A consistência nos comandos de manipulação da SQL em relação a um atributo das *variáveis tabela* (exemplo: se, no comando UPDATE, o programador permitisse uma alteração de um atributo que é chave estrangeira de outra tabela – acusaria o erro de consistência);
- Se a segurança das transações entre os comandos de manipulação da base de dados está correta (exemplo: um comando que removesse uma tupla que tem uma regra relacionada à outra tabela – o sistema não deveria permitir a exclusão da tupla).

1.2.1 – Teste de Integração em uma ABDR

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 os métodos (UPs) são ocasionadas pelas variáveis globais ou por variáveis passadas por parâmetros através de comandos de chamadas (SPOTO, 2000). 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 (SPOTO, 2000).

A dependência dos dados supramencionada pode ser dividida em 2 grupos: dependência interna ou *Intra-classe e* dependência externa ou *Inter-classe.*

- **Dependência Interna ou *Intra-classe***: ocorre quando existe 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. Nas Figuras 2.2 e 2.3 (BECARI, 2005), são mostrados dois exemplos de dependência *Intra-classe*.

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. Podem ser divididos em todos os ***t-usos-ciclo1-intra*** e todos os ***t-usos-ciclo2-intra*** (SPOTO, *et al.*, 2005).

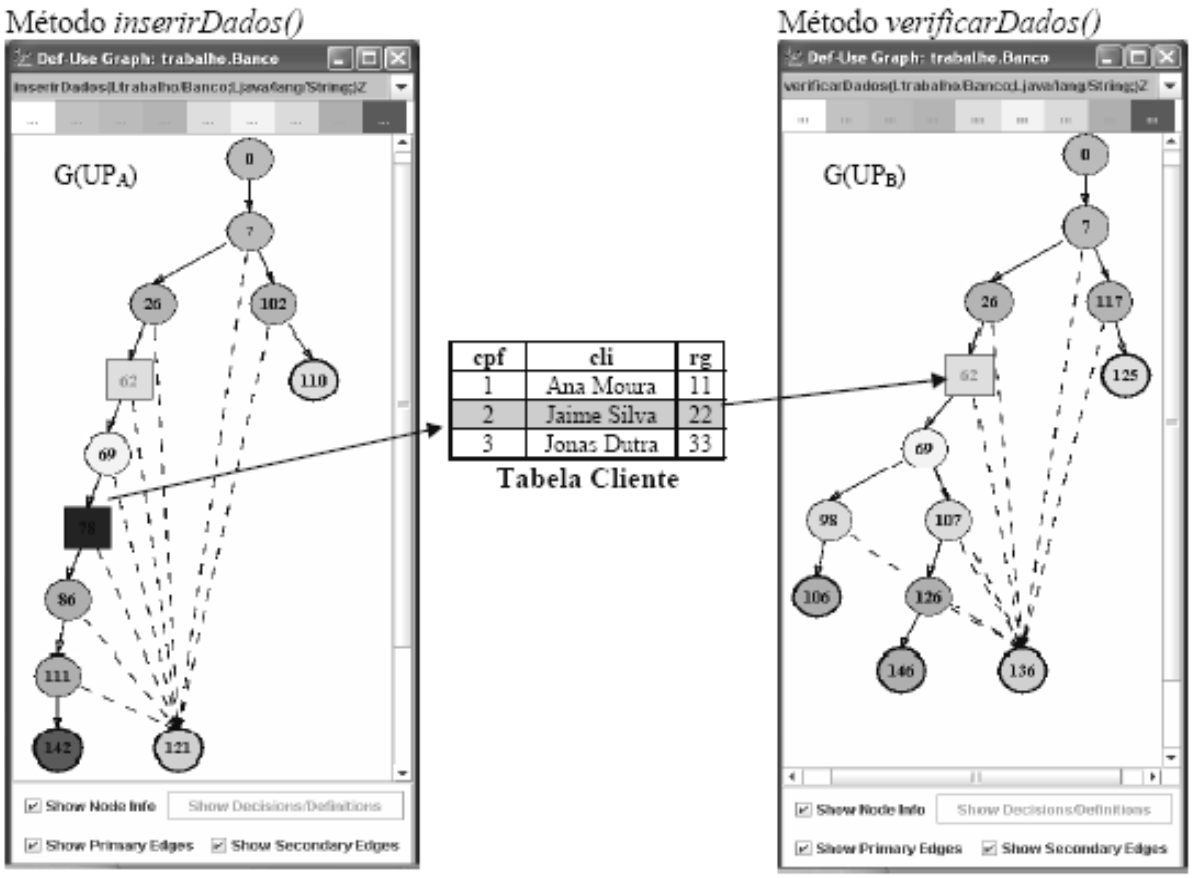

Figura 1.2 - Dependência de Dados entre dois Métodos de uma mesma Classe - Ciclo 1.

Todos os t-usos-ciclo1-intra: neste critério utiliza-se somente uma variável tabela ***t***, podendo ser estabelecida uma associação contendo duas unidades, uma de definição e outra de uso, mas para que a associação seja efetivada, tanto a definição persistente de ***t*** quanto o uso de ***t***, deve ocorrer na mesma tupla da tabela. Na Figura 1.2 (BECARI, 2005), é mostrado um exemplo onde se aplica o critério todos os *t-usos-ciclo1-intra.*

Todos os t-usos-ciclo2-intra: neste critério é necessária a execução de outra unidade para satisfazer a associação, além das unidades de definição e uso. Isso ocorre quando existem dependências múltiplas, de maneira que para definir a variável ***t*** seja necessário definir a variável ***t'*** e, para isso, pode ser necessário à execução da variável que define ***t'*** para depois executar a unidade que define ***t***, e finalmente, executar o método que usa ***t***. Na Figura 1.3 (BECARI, 2005), é mostrado um exemplo onde se aplica o critério todos os *t-usos*-ciclo2-intra.

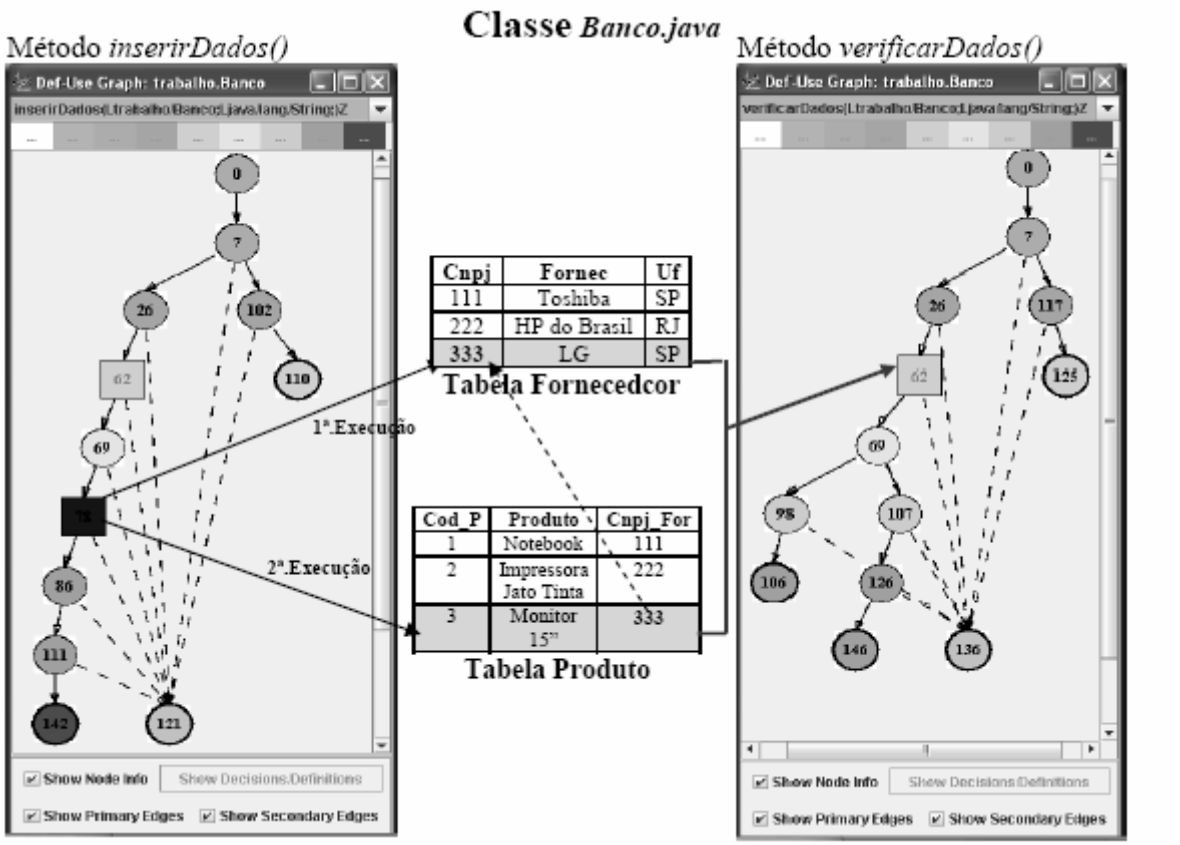

Figura 1.3 – Dependência de Dados entre dois Métodos de uma mesma Classe – Ciclo 2.

- **Dependência Externa ou *Inter-classe***: ocorre quando existe uma dependência de dados entre dois ou mais métodos de classes diferentes com relação a variável tabela, podendo estar relacionada a uma ou mais variáveis tabelas. A tabela deve estar disponível para as duas classes. Na Figura 1.4 (BECARI, 2005), é mostrado um exemplo de dependência *Inter-classe*.

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. Isso visa a 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. Considerando dois métodos UP_A, pertencente a uma classe Mod_x, e o método UP_B, pertencente a uma outra classe Mod_y, tem-se:

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;

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

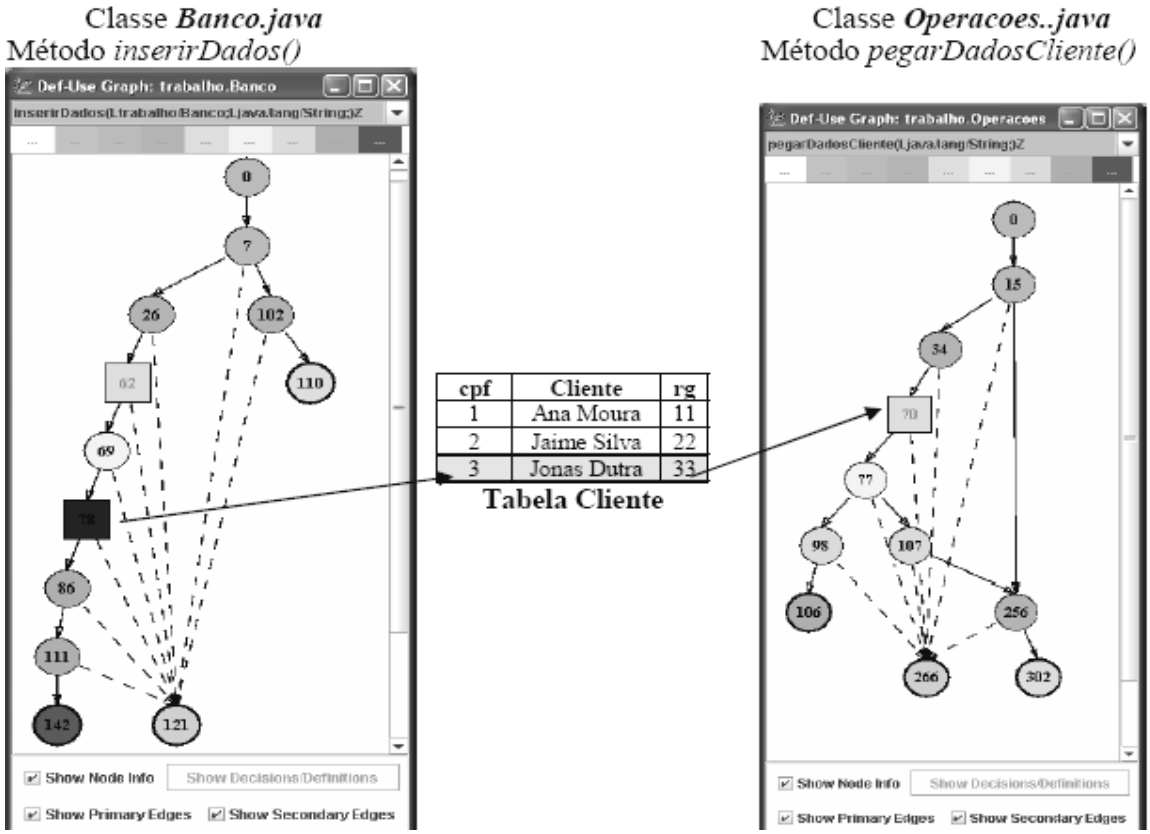

Figura 1.4 – Dependência de Dados entre dois Métodos de Classes Diferentes.

Através dos critérios inter-classe propostos em Spoto, *et al.* (2005), pode-se verificar na Figura 1.4, a existência do método *InserirDados()* pertencente à classe Banco.java e do 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 **t**-uso ao selecionar a mesma tupla com CPF=3, no arco (70, 77), na mesma tupla que foi definida.

Nota-se também na Figura 1.4, 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>. Nesta 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 executado pela mesma tupla.

Talvez com a execução de apenas duas unidades (uma de definição e outra de uso), algumas dependências não podem ser exercitadas, para isso 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, executar o método que usa ***t*** (BECARI, 2005).

Quando os métodos de uma mesma classe possuem apenas comandos que caracterizam ocorrências de uso de tabelas, não é possível a geração da integração *Intra-classe* para exercitar as dependências existentes nos métodos desta classe (classes que só executam relatórios, por exemplo). Nesse caso, a integração *Inter-classe* (integram métodos de classes distintas) complementa a etapa de teste exercitando tais métodos cujos comandos possuem apenas *Queries,* integrando, assim, com classes que possuem métodos com definição persistente.

Os critérios de teste estrutural em ABDR contribuem, assim, para melhorar as chances de escolha dos dados de teste para variáveis persistentes visando a detectar determinados tipos de defeitos relacionados ao modelo relacional (BECARI, 2005).

Defeitos em comandos de consultas da SQL (uso de "*Queries*" indevidas) são detectados ao exercitar os elementos requeridos pelos critérios de integração (*Intra-modular* e *Inter-modular*) (SPOTO, 2000).

2 – PROCEDIMENTOS DE EXECUÇÃO DE TESTE DE *SOFTWARE*

2.1 – Normas de Testes

Neste Capítulo serão apresentados conceitos sobre Normas de teste, bem como sua estruturação e organização, com intuito de reger a elaboração e execução de testes de um *Software*.

As Normas de testes dão suporte aos engenheiros de *Software*, no sentido de organizar o processo de teste, possibilitando a criação de uma metodologia de teste (CRESPO, *et al.*, 2004). A metodologia está fundamentada na adoção de um processo de teste e nos artefatos sugeridos pela Norma IEEE 829-1998 (IEEE-Std-829, 1998). Uma metodologia de teste pode ser aplicada a qualquer tipo de *Software*, seja sistema de informação ou *Software* científico.

A implantação do processo de teste com base numa metodologia envolve um conjunto de atividades, que vai desde o levantamento das necessidades da empresa, passa pela realização de treinamentos da equipe técnica e vai até ao acompanhamento dos trabalhos realizados, formando assim um completo ciclo de implantação dentro de uma empresa, como é mostrado na Figura 2.1 (CRESPO, *et al.*, 2004).

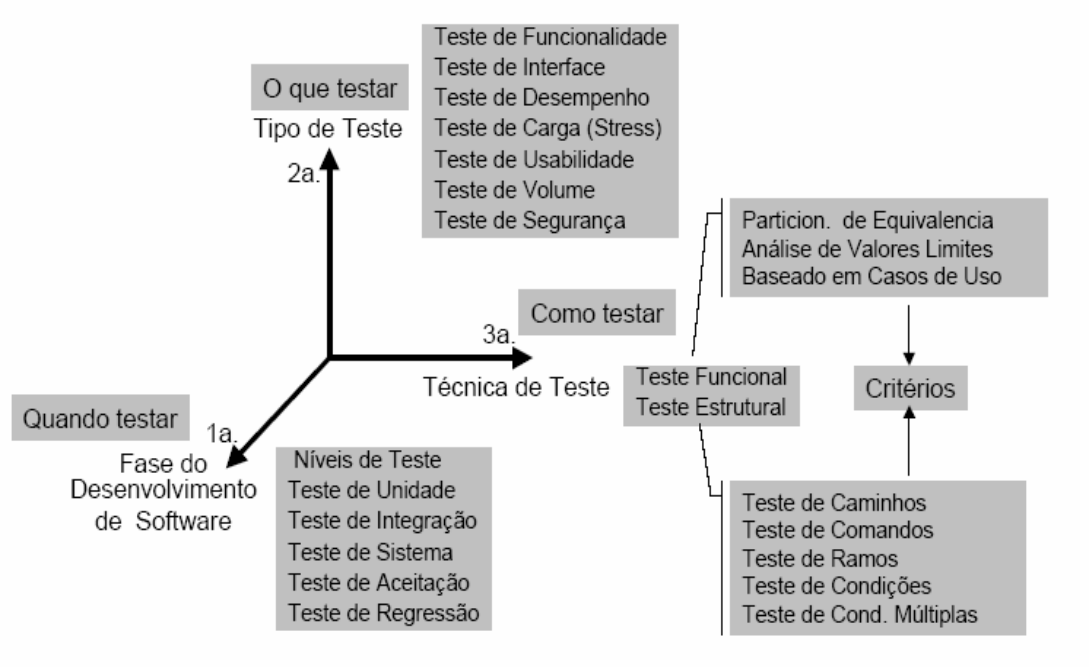

Figura 2.1 – Relação entre níveis, tipos e Técnicas de Teste.

A metodologia de teste pode ser dividida em 3 partes: treinamento, processo de teste e suporte para geração de documentos.

1 – Treinamento: é aplicada por meio de cursos e consiste na capacitação em conceitos básicos sobre teste de *Software*, técnicas de teste, documentação de teste e processo de teste.

2 – Processo de Teste: a metodologia define um processo genérico de teste que prevê a realização das atividades de planejamento, projeto, execução e acompanhamento dos testes de unidade, integração, sistemas e aceitação.

3 – Suporte para Geração de Documentos: aplica-se uma técnica para criação de documentos, com base na Norma IEEE 829-1998, que serão utilizados para a gerência do processo de teste.

Uma premissa básica da metodologia de teste proposta, é que o processo de teste, quando adequadamente definido, pode ter um impacto positivo nos resultados de diversas outras atividades de desenvolvimento. Desta forma, o enfoque das atividades de teste não é

somente identificar problemas, mas principalmente prevenir problemas. Estas premissas estão presentes em diversas referências sobre teste preventivo (BEIZER, 1995; CRAIG, 2002).

O fato do processo genérico proposto não contemplar a automação de teste reflete a visão de que um processo só deve ser suportado por ferramentas quando estiver convenientemente definido e consistentemente adotado.

Segundo Zallar (2001), o processo de automação de teste tem maior probabilidade de ser bem sucedido para organizações que possuam uma equipe de teste bem definida e com um processo padrão de documentação seguido.

2.1.1 – Aspectos Gerais da Norma IEEE 829-1998

A Norma IEEE 829-1998 descreve um conjunto de documentos para as atividades de teste de um produto de *Software*. Serão representados a seguir, 8 documentos que são definidos pela Norma para cobrir as tarefas de planejamento, especificação e relato de testes (CRESPO, *et al.*, 2004).

Plano de Teste: apresenta o planejamento para execução do teste, incluindo a abrangência, abordagem, recursos e cronograma das atividades de teste. Identificar os itens e as funcionalidades a serem testados, as tarefas a serem realizadas e os riscos associados com a atividade de teste.

A tarefa de especificação de testes é abrangida pelos 3 documentos seguintes.

Especificação de Projeto de Teste: refina a abordagem apresentada no plano de teste e identifica as funcionalidades e características a serem testadas pelo projeto e por seus testes associados. Este documento também identifica os casos e os procedimentos de teste, se existir, e apresenta os critérios de aprovação.

Especificação de Caso de Teste: define os casos de teste, incluindo dados de entrada, resultados esperados, ações e condições gerais para a execução do teste.

Especificação de Procedimento de Teste: especifica os passos para executar um conjunto de casos de teste.

Os relatórios de teste são cobertos pelos 4 documentos seguintes.

Diário de Teste: apresenta registros cronológicos dos detalhes relevantes relacionados com a execução dos testes.

Relatório de Incidente de Teste: documenta qualquer evento que ocorra durante a atividade de teste e que requeira análise posterior.

Relatório – Resumo de Teste: apresenta de forma resumida os resultados das atividades de teste associadas com uma ou mais especificações de projeto de teste e provê avaliações baseadas nesses resultados.

Relatório de Encaminhamento de Item de Teste: identifica os itens encaminhados para teste no caso de equipes distintas serem responsáveis pelas tarefas de desenvolvimento e de teste.

A Norma IEEE 829-1998 divide as atividades de teste em três etapas: preparação do teste, execução do teste e registro de teste. Na Figura 2.2 (CRESPO, *et al.*, 2004), são mostrados os documentos produzidos por cada uma dessas etapas e os relacionamentos entre eles.

Segundo Crespo, *et al.* (2004), os 8 documentos referentes às atividades de teste definidos pela Norma IEEE-std-829-1998, podem ser comparados com 3 documentos propostos pela Norma ISO/IEC 12207 (ISO/IEC, 1998), que define uma estrutura comum para os processos do ciclo de vida de *Software*. Na Tabela 2.1 segue a demonstração dessa comparação.

IEEE 829-1998	ISO/IEC 12207
Plano de Teste.	Plano de Teste
Especificação de Projeto de Teste, Especificação de Caso de Teste, Especificação de Procedimento de Teste.	Procedimento de Teste
Relatório de Encaminhamento de Item de Teste, Relatório de Incidente de Teste, Diário de Teste, Relatório-Resumo de Teste.	Relatório de Teste

Tabela 2.1 – Comparação de Documentos entre as Normas IEEE 829-1998 e ISO/IEC-12207.

Phillips (1998), sugere que o teste de *Software* em sistemas que não sejam extremamente grandes ou complexos, utilize apenas 3 documentos da Norma IEEE-std-829-1998: plano de teste, especificação de procedimento de teste e relatório-resumo de teste. Alguns documentos como relatório de encaminhamento de item de teste, diário de teste e relatório de incidente de teste, podem ser implementados através de formulários.

Mais do que apresentar um conjunto de documentos que deva ser utilizado ou adaptado para determinadas empresas ou projetos, a Norma apresenta um conjunto de informações necessárias para o teste de produtos de *Software*. A correta utilização da Norma auxiliará a gerência a se concentrar tanto com as fases de planejamento e projeto quanto com a fase de realização de testes propriamente dita, evitando a perigosa armadilha de só iniciar a pensar no teste de um produto de *Software* após a conclusão da fase de codificação (CRESPO, *et al.*, 2004).

A Norma IEEE 829-1998 pode ser aplicada também em projetos pequenos ou de baixa complexidade, para isso os documentos propostos podem ser agrupados, diminuindo assim, o gerenciamento e custo de produção dos documentos. Neste último caso, os documentos podem ainda ter seus documentos abreviados.

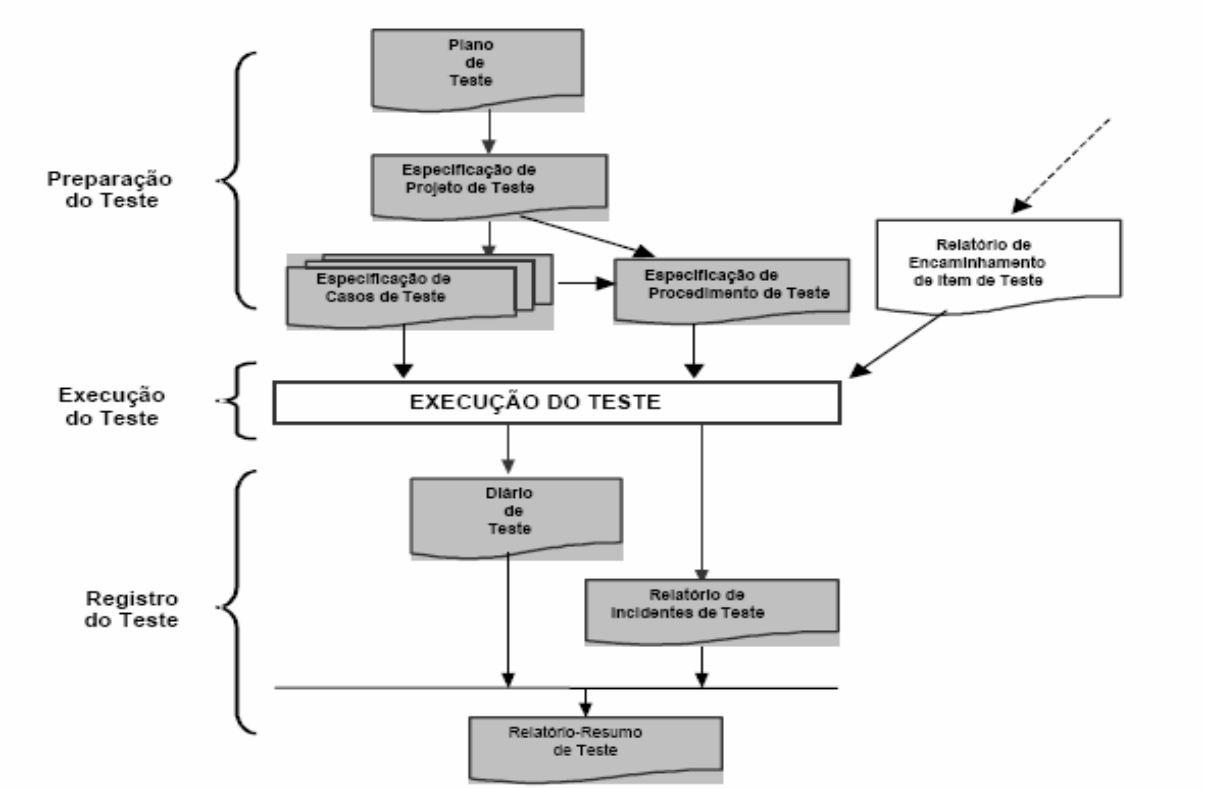

Figura 2.2 – Relacionamento entre os Documentos de Teste.

Além de apresentar um conjunto de documentos – que deve ser adaptado ou utilizado em determinadas empresas ou projetos – a Norma IEEE 829-1998 apresenta um conjunto de informações necessárias para o teste de produtos de *Software*.

A correta aplicação e utilização da Norma IEEE 829-1998, auxiliará a gerência a se concentrar tanto nas fases de planejamento e projeto quanto nas fases de realização de teste propriamente dita, evitando assim o início da fase de teste após a conclusão da fase de codificação (CRESPO, *et al.*, 2004).

Os processos abrangem a preparação, a execução e o registro dos resultados do teste, esses processos estabelecem uma orientação geral e, se necessário, podem ser modificados para adequar-se às situações particulares de organizações envolvidas nas atividades de teste. Um processo é definido para cada documento da Norma, segundo a seguinte estrutura:

1. Funções e responsabilidades no processo – participantes na execução das tarefas;

2. Critérios para o início do processo – elementos e/ou condições necessários para iniciar a execução das tarefas;

3. Entradas do processo – dados, recursos ou ferramentas necessários para a execução das tarefas;

4. Tarefas do processo – ações necessárias para produzir as saídas do processo. Para cada tarefa são identificadas suas entradas, com indicação de possíveis fontes, e as saídas produzidas. A ordem de apresentação das tarefas não reflete necessariamente a seqüência em que devem ser executadas;

5. Saídas do processo – dados ou produtos gerados pela execução das tarefas;

6. Critérios para término do processo – elementos e/ou condições necessários para encerrar a execução das tarefas;

7. Medições do processo – medidas a serem coletadas como parte da execução das tarefas.

Dependendo do domínio da aplicação, da estratégia ou da fase de teste, os processos podem ser adaptados de modo a produzir um conjunto maior ou menor de documento, mas independentemente de qualquer fator, os documentos de preparação para o teste devem incluir: planejamento de teste, projeto de teste, casos de teste e procedimentos de teste.

Caso seja necessário, os passos ou tarefas dos processos podem ser estendidos, com intuito de incluir ações adicionais, que podem eventualmente, resultar em novos documentos e/ou formulários.

3 – CONCEITOS DE *WORKFLOW*

Pode-se definir *Workflow* como: **qualquer tarefa** executada em **série ou em paralelo** por **dois ou mais membros** de um grupo de trabalho (*Workgroup*) visando um **objetivo comum** (MORO, 1998). De acordo com a definição mencionada, é interessante explicitar que:

- **Qualquer Tarefa**: implica que o *Workflow* se refere a um largo campo de atividades.
- **Em Série ou Paralelo**: isso implica que os passos na tarefa podem ser executados um depois do outro ou simultaneamente por diferentes indivíduos, ou pela combinação dos dois.
- **Dois ou mais Membros**: implica que se somente uma pessoa executar uma tarefa isso não é *Workflow*. Como o nome *Workflow* sugere, uma tarefa é um *Workflow* se "flui" de um indivíduo para outro.
- **Objetivo Comum**: indivíduos participando em um *Workflow* devem estar trabalhando em busca de um único objetivo. Trabalhar em projetos independentes não constitui um *Workflow*.

De uma forma simplificada, pode-se definir um *Workflow* como, uma coleção de tarefas organizadas para realizar um processo, quase sempre de negócio. Essas tarefas podem ser executadas por um ou mais sistemas de computador, por um ou mais agentes humanos, ou então por uma combinação destes. A ordem de execução e as condições, pelas quais, cada tarefa é iniciada também estão definidas no *Workflow*, sendo que o mesmo é capaz ainda de representar a sincronização das tarefas e o fluxo de informações (MORO, 1998).

3.1 – Categorias de *Workflow*

Os produtos de *Workflow* podem ser divididos em três categorias gerais (ATLEE, 1997):

- ***Document Routing*:** estabelece fluxo de informação e faz o roteamento dos mesmos;
- ***Ad-hoc*:** Ferramentas de G*roupware,* não existe uma estrutura pré-definida para o processo, ou esta estrutura pode ser modificada em tempo de execução. Fornece o gerenciamento de *Workflow* através de *Templates* ou formulários baseados em mensagens. O fluxo é feito pelo servidor de roteamento de mensagens. Exemplos: criação de documentos, desenvolvimento de *Software*, requisições de viagem, campanha de *marketing* para lançamento de produto;
- **Automação de Processo de Negócios:** sistema para definir processos de negócios e implementação dos mesmos, em *Software*.

3.2 – Criação de um *Workflow*

De acordo com Moro (1998), um processo de *Workflow* pode ser criado através dos seguintes passos:

- Define-se uma atividade ou tarefa que um grupo de trabalho precisa realizar e as regras de serviço que gerenciarão a atividade;
- Divide-se a tarefa em sub-tarefas (passos). Cada passo representa uma lista bem definida de coisas que são realizadas por um indivíduo e que são feitas logicamente juntas. Uma tarefa pode ser quebrada em passos de maneiras

diferentes. Nesse ponto, é exigido um julgamento do serviço para decidir onde dividir uma tarefa;

- Decide-se o conjunto de habilidades para realizar cada passo. Isso irá especificar as funções ou indivíduos de trabalho que podem ser chamados para realizar tal passo;
- Decide-se a seqüência em que cada passo deve ser realizado;
- Se algum dos passos é realizado em uma base condicional, identificam-se esses passos e definem-se as condições;
- Projeta-se um mapa do *Workflow* que identifica os passos e a seqüência, ou "fluxo" em que os passos devem ser realizados. Associam-se funções ou indivíduos de trabalho a cada passo;
- Cria-se os formulários, documentos e instruções que serão usados pelos indivíduos em cada passo para execução da sub-tarefa.

Pode-se perceber que, um *Workflow* envolve uma seqüência ou passos ou um processo. A tarefa "flui" de um passo para outro baseado em regras e condições pré-definidas.

3.3 – Exemplo de *Workflow*

Na Figura 3.1 (MORO, 1998), é apresentado um exemplo de *Workflow* que representa um sistema de suporte “on-line” para usuários. Este *Workflow*, como pode ser visto, executa acesso a bases de dados, armazenando os dados gerados em uma atividade para o uso nas atividades seguintes.

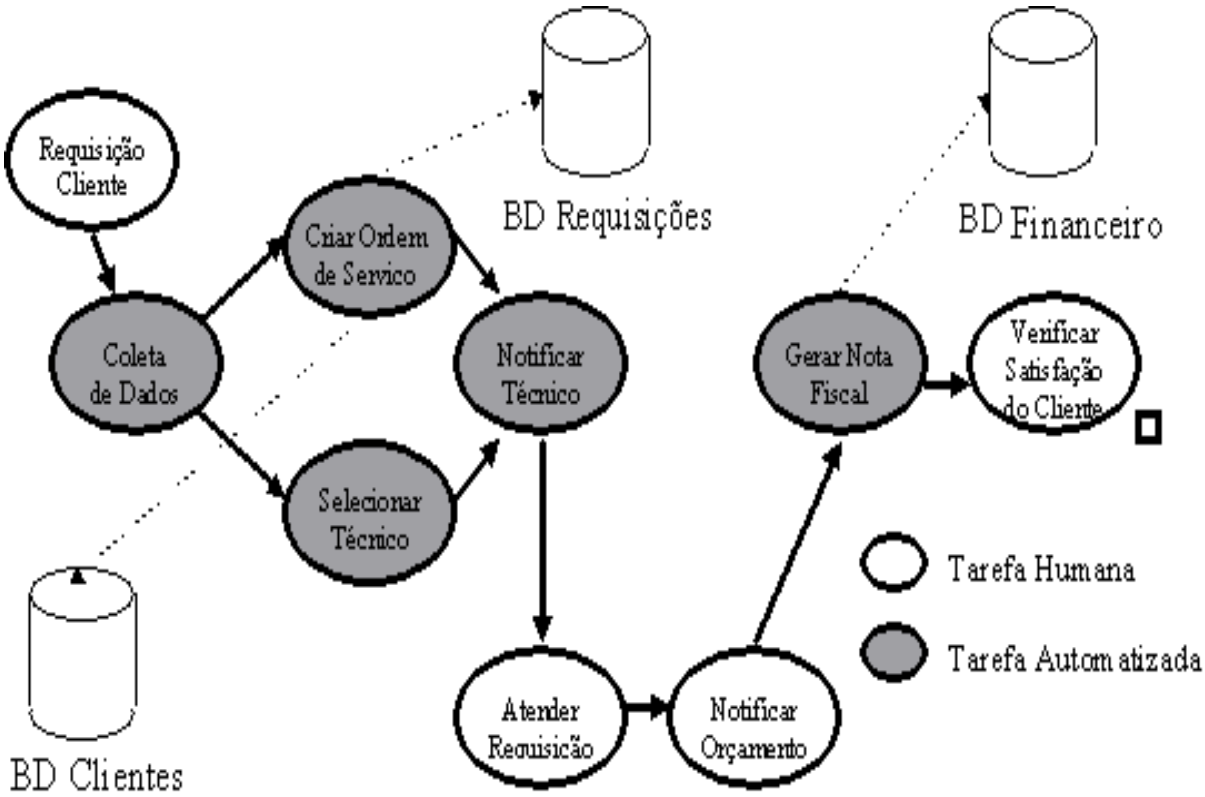

Figura 3.1 - Exemplo de *Workflow* de um Sistema de Atendimento On-line.

Segue algumas descrições de mais alguns exemplos de *Workflow*:

- Sistema de recursos humanos que manuseiam empregados ou processos de transferência de trabalhos internos, coordenando as reclamações dos empregadores, gerenciando a partida de empregados e assim por diante;
- Gerenciamento de ciclos de vendas e processamento de ordem on-line de compras;
- Reenvio de informações que tenham sido passadas por fax, gravadas em fitas ou necessitam de intervenção manual para ligar dois formatos diferentes;

3.4 – *Workflow* para Testes de *Software*

Os *Workflows* para testes modelam os processos de testes de *Software*, mas são poucos os trabalhos que comentam a existência de *Workflows* específicos para o teste de sistema. Os testes têm sido alvos de estudos e evolução, em função do impacto causado no nível de qualidade do *Software* final (MATOS, 2004).

O desenvolvimento de técnicas, que automatizam o processo de teste e principalmente controlam mais efetivamente o processo de teste, possibilitam resultados mais criteriosos e, com mais qualidade (MATOS, 2004).

3.4.1 – *Workflow* de Testes do RUP

O RUP oferece uma metodologia de projeto de *Software*, no qual, são descritos métodos para o desenvolvimento de *Software* usando técnicas testadas anteriormente. O RUP consiste de um processo pesado, podendo desta forma, ser aplicado a grandes equipes de desenvolvimento, onde a necessidade e a dificuldade de gerenciamento são maiores (MATOS, 2004).

O RUP é um processo analítico, incremental e iterativo, baseado em ciclos, onde ao final de cada ciclo haverá um produto de *Software*, o processo segue as seguintes fases: iniciação, elaboração, construção e transição. Na Figura 3.2 (MATOS, 2004), é ilustrado o *Workflow* para teste de *Software* do RUP.

O RUP possui um *Workflow* para cada uma das etapas do ciclo de desenvolvimento, cujos passos são: modelagem do negócio, elicitação e análise dos requisitos de *Software*, análise e projeto, implementação, teste, implantação, gerência de configuração e de mudanças, gerência do projeto e gerência do ambiente (MATOS, 2004).

O *Workflow* para testes do RUP utiliza-se basicamente de duas técnicas:

- **Plano de Teste:** técnica que enfoca planejar todo o processo de testes, incluindo análises de riscos e definição de responsabilidades.
- **Caso de Teste:** conjunto de dados desenvolvido para um teste específico, possuindo condições de execução e resultados esperados.

A atividade de teste no RUP pode ser vista sob três dimensões diferentes: qualidade, estágio de teste e tipo de teste.

- **Qualidade:** os aspectos de confiabilidade e desempenho precisam ser assegurados.
- **Estágio de Teste:** sugere a implementação da atividade de teste em etapas distintas e progressivas. O RUP referencia quatro diferentes estágios de teste: teste de unidade (teste individual de pequenas porções do *Software*), teste de integração (teste de componentes individuais ou subsistemas), teste de sistema (teste do sistema por completo) e teste de aceitação (teste do *Software* realizado pelos usuários finais).
- **Tipo de Teste:** são subdivididos em quatro tipos: teste de *benchmark* (compara os objetivos de teste com os padrões conhecidos), teste de integridade (verifica confiabilidade, robustez e tolerância a falhas), teste de performance (testa a

performance do teste em diferentes configurações), e teste de estresse (teste da performance em condições anormais ou extremas).

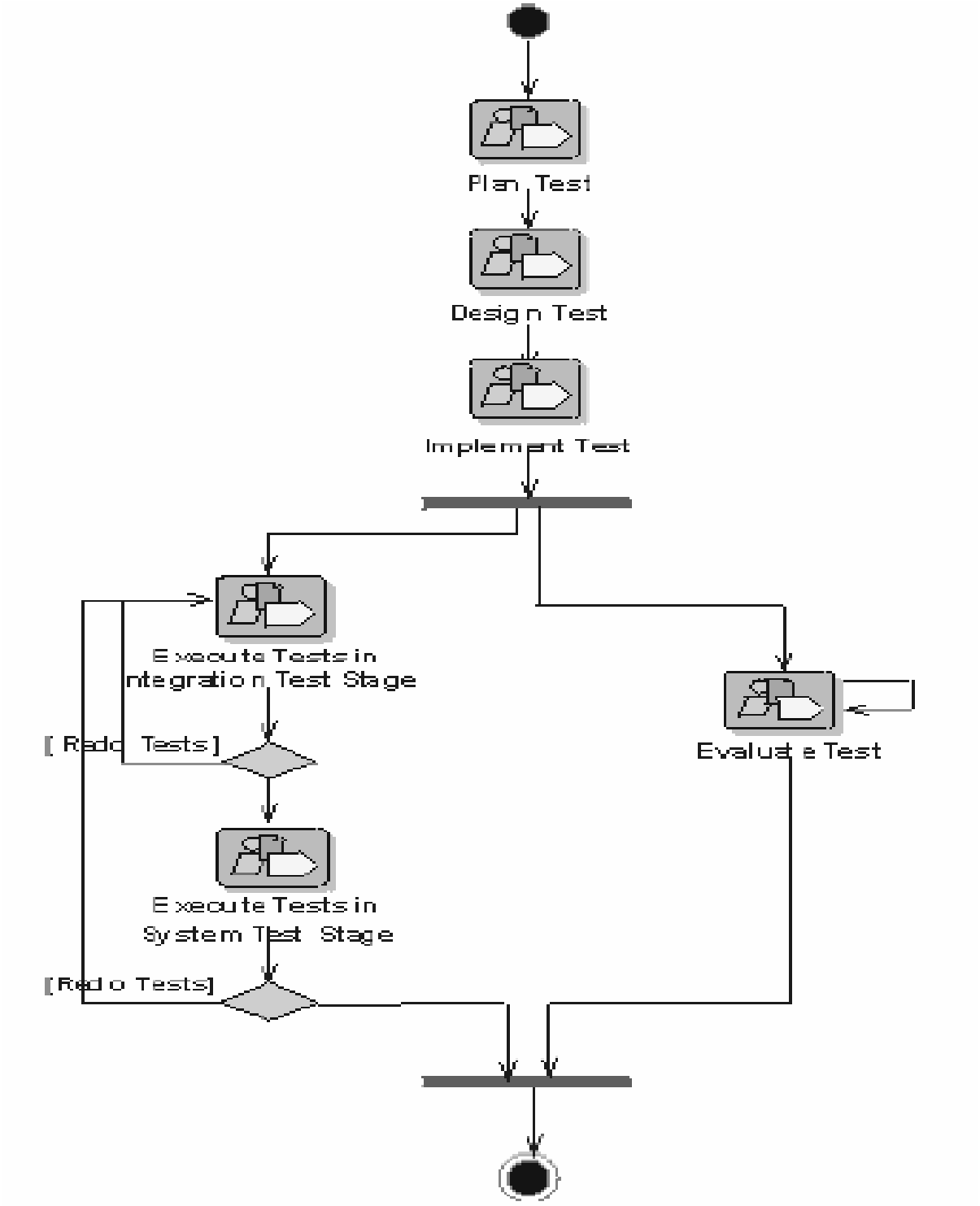

Figura 3.2 – *Workflow* para Testes de *Software* Original do RUP.

3.4.2 – *Workflow* de Teste da Ferramenta FADAT

Segundo Parckert (2006), analisando as seguintes perspectivas sobre teste de *Software*: tempo, alto custo e a escassez de ferramentas; foi que surgiu o interesse em desenvolver uma ferramenta para documentar e apoiar a aplicação de *Workflows* de teste, a qual tem como principal objetivo facilitar a atividade de documentação de teste de *Software* para os testadores.

Na Figura 3.3 (PARCKERT, 2006), é representado o *Workflow* criado e adotado para controlar as atividades da FADAT, sendo que este foi adaptado a partir do *Workflow* de testes do RUP (KRUCHTEN, 2000). O *Workflow* adotado para a ferramenta é composto das cinco atividades seguintes: planejar teste, projetar teste, implementar teste, executar teste e avaliar teste, deve-se salientar que as atividades de executar teste e avaliar teste podem ocorrer de forma paralela.

Caso a atividade executar teste tenha que ser refeita, faz-se necessário criar uma nova versão para esta atividade, de tal forma que a versão criada anteriormente não deve ser removida.

Através do diagrama de classe apresentado na Figura 3.4 (PARCKERT, 2006), pode-se observar que:

A classe *Usuário* é responsável por armazenar todos os dados relevantes dos usuários, sendo que eles poderão exercer diferentes papéis na ferramenta e, o atributo *Permissão_User* da classe *Usuário* determinará qual classe pertencerá um usuário.

A classe *Workflow* gerencia todos os dados relevantes a um determinado projeto de *Workflow*, sendo possível a criação, atualização, consulta e remoção de um projeto de *Workflow*. Cada *Workflow* é associado a um usuário, que será o usuário proprietário do *Workflow*, e a uma ou várias atividades.

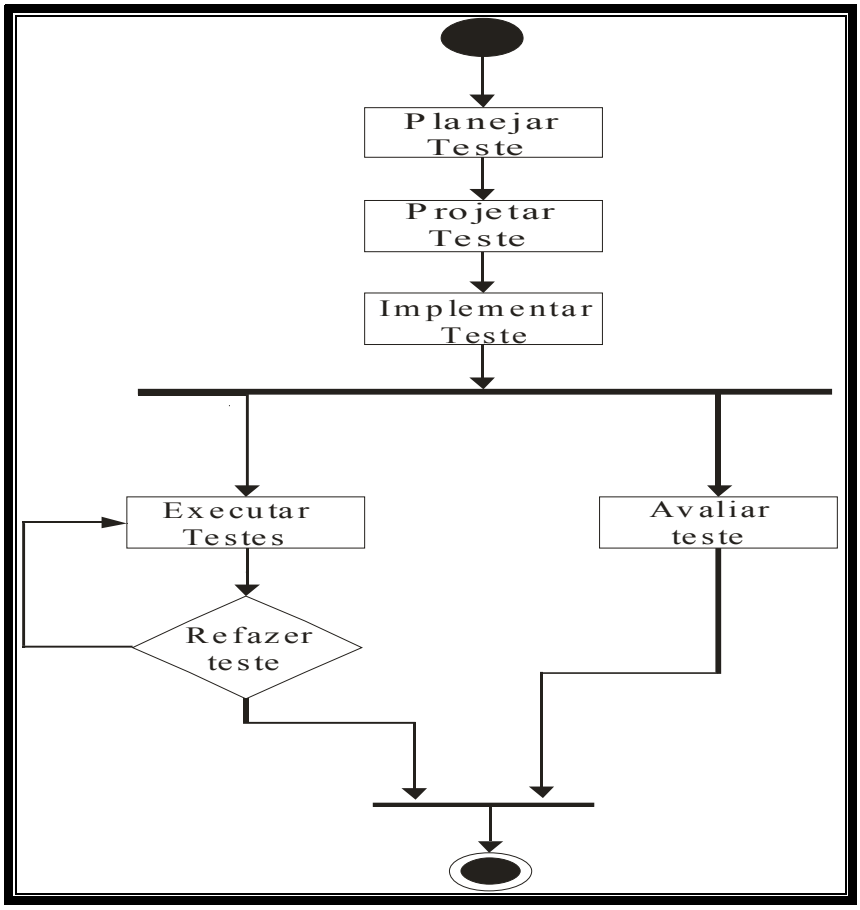

Figura 3.3 – *Workflow* de Teste adaptado de Kruchten (2000).

A classe *Atividade* é responsável por controlar os dados referentes a cada atividade, de tal forma que, as atividades serão padronizadas para todos os projetos de *Workflow*.

A classe *Executor* é responsável por controlar os dados referentes a cada executor, mas o executor deverá estar devidamente cadastrado na ferramenta, pois para cada atividade haverá um executor.

A classe *Ferramenta_Apoio* armazenará os dados referentes a uma ferramenta de apoio, que por sua vez, poderá auxiliar uma determinada atividade de teste. A classe *Fase_Teste* terá o objetivo de controlar quais as fases de teste que uma determinada ferramenta abrange.

Uma atividade atende a uma técnica de teste, sendo que a classe *Técnica_Teste* será responsável por armazenar qual a técnica de teste que uma determinada atividade atende. A

classe *Técnica_Teste* possui um critério de teste, de tal forma que, a classe *Critério_Teste* será responsável por armazenar quais os critérios que uma determinada técnica de teste possui.

Durante a execução das atividades serão produzidos artefatos de entrada ou de saída, de tal forma que, determinadas atividades necessitarão de artefatos de entrada para produzirem artefatos de saída, para gerenciar este controle foram criadas as classes *Artefato* e *Artefato_Atividade.*

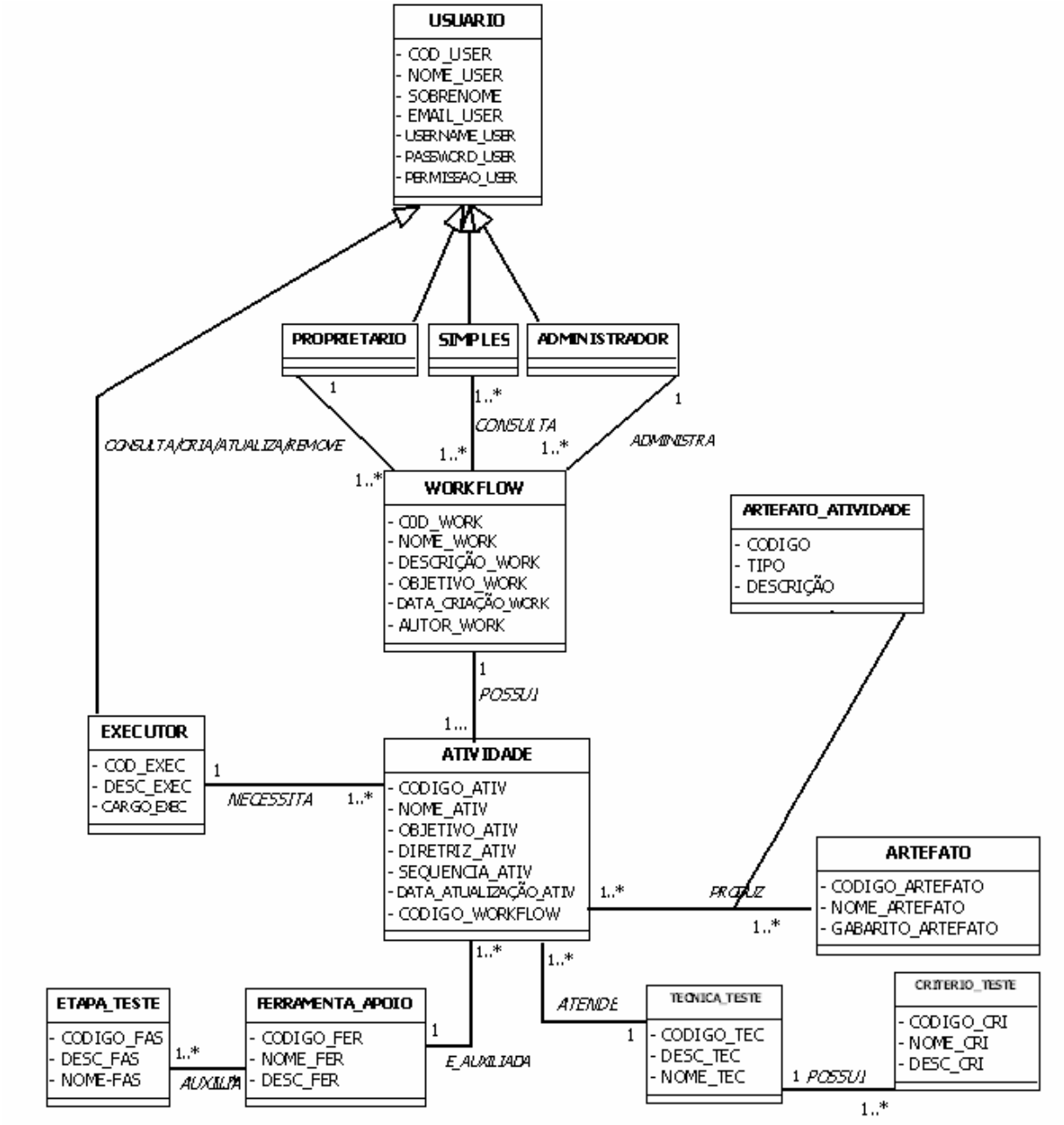

Figura 3.4 – Diagrama de Classe da FADAT.

Com base na documentação proposta pela Norma IEEE-std-829-1998 e no modelo conceitual da FADAT (diagrama de classe) representado na Figura 3.4 (PARCKERT, 2006), identificou-se as seguintes funcionalidades para a implementação desta ferramenta:

- **Login na Ferramenta:** libera o acesso à visualização e cadastros diversos, após informar-se um nome de usuário e sua respectiva senha;
- **Gerenciamento de Usuários:** controla o gerenciamento de usuários na ferramenta, através das funções de cadastro, alteração e remoção de usuários da ferramenta;
- **Gerenciamento de *Workflow*:** gerencia o cadastro, alteração e remoção de projetos de *Workflows* na ferramenta;
- **Gerenciamento de Atividades:** gerencia as atividades referentes a um determinado *Workflow,* através das funções de cadastro, alteração e remoção de atividades;
- **Gerenciamento de Executores:** determina os executores que realizaram as atividades referentes a um determinado projeto de *Workflow,* permite a inclusão, alteração de executores;
- **Gerenciamento de Ferramentas de Apoio:** controla as ferramentas de Apoio que poderão ser utilizadas para uma determinada atividade de teste de *Software*, esta funcionalidade é complementada pela funcionalidade gerenciamento de fase de teste;
- **Gerenciamento de Fases de Teste:** complementa a funcionalidade de gerenciamento de ferramentas de apoio*,* esta funcionalidade irá gerenciar quais as fases de teste que são cobertas por uma ferramenta;

- **Gerenciamento de Técnicas de Teste:** gerencia quais as técnicas de teste que uma determinada atividade atende, esta funcionalidade é complementada pela funcionalidade gerenciamento de critérios de teste;
- **Gerenciamento de Critérios de Teste:** complementa a funcionalidade gerenciamento de técnicas de teste, definindo quais os critérios de teste que determinada técnica de teste possui;
- **Gerenciamento de Artefatos:** define os artefatos que serão gerados e utilizados durante a execução das atividades, permitindo a inclusão, alteração e remoção dos artefatos;
- **Consultas Diversas:** permite que o usuário realize as consultas referentes às funcionalidades descritas anteriormente.

Através da arquitetura representada na Figura 3.5 (PARCKERT, 2006), pode-se observar a existência de quatro módulos, os quais podem se comunicar entre si, que geram artefatos referentes à documentação da atividade de teste de *Software*, tendo como base a Norma IEEE-std-829-1998.

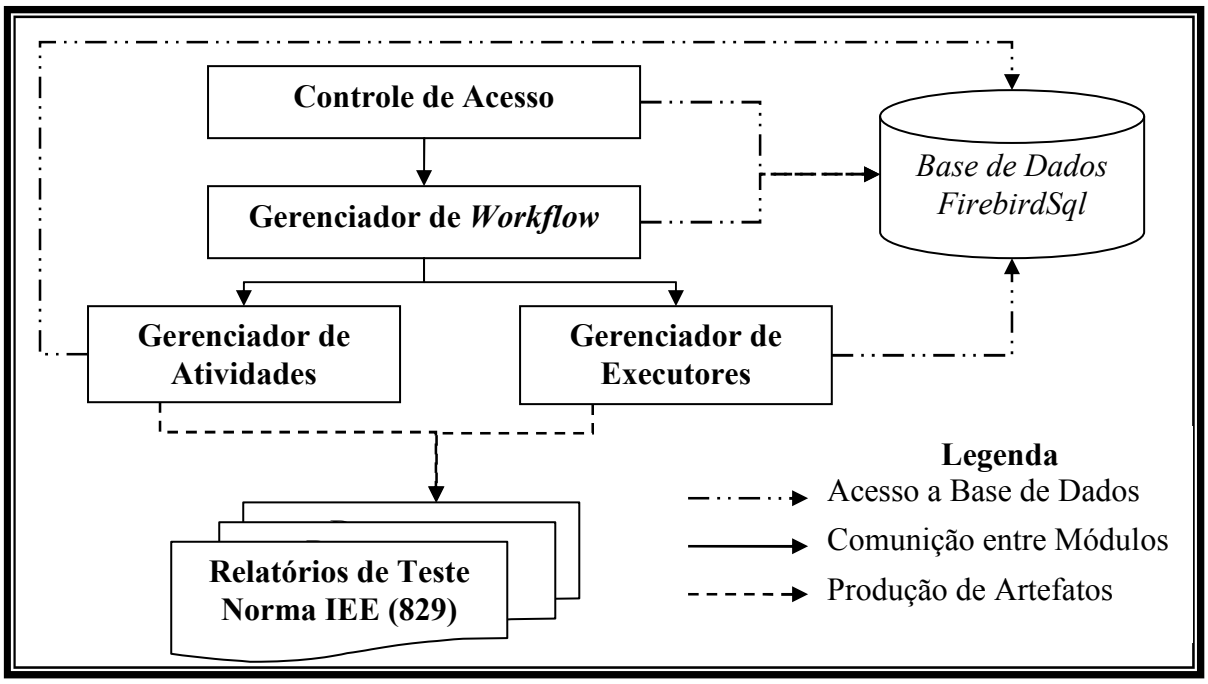

Figura 3.5 – Arquitetura da FADAT.

- **Controle de Acesso:** este módulo é responsável por validar o acesso às funcionalidades da ferramenta ao usuário, isto ocorrerá após o usuário informar seu nome de usuário e sua respectiva senha, após esta verificação será liberado o acesso às funcionalidades;
- **Gerenciador de *Workflow*:** este módulo tem como principal objetivo gerenciar os *Workflows* cadastrados na ferramenta, permitindo a consulta, inserção, alteração e remoção dos *Workflows*, sendo que a última só poderá ser realizada se o *Workflow* não tiver nenhuma dependência;
- **Gerenciador de Executores:** este módulo tem como finalidade principal gerenciar quais serão os executores responsáveis por cada atividade referente a um determinado *Workflow*, podendo ser executada somente pelo autor do *Workflow,* permitindo a inclusão, alteração e consulta de usuário por atividade;
- **Gerenciador de Atividades:** este módulo é responsável por gerenciar as atividades que deverão ser realizadas, com intuito de produzir um relatório de teste de *Software*, estas atividades estão diretamente ligadas ao *Workflow* adotado na Figura 3.3 (PARCKERT, 2006).

4 – *WORKFLOW* DE TESTE DE ABDR

Neste Capítulo será apresentado um *Workflow* de teste voltado para ABDR, onde serão demonstrados os passos necessários para o seu desenvolvimento, cuja abordagem principal da construção deste *Workflow,* está referenciada à aplicação da técnica de teste estrutural analisando o fluxo de dados em ABDR, visando o nível teste de unidade.

Paralelamente ao desenvolvimento do *Workflow* supramencionado, serão apresentados os resultados com base em um exemplo – no qual foi utilizado um código para a realização de um estudo de caso, apresentado no (APÊNDICE A) – a fim de que se possam ser exemplificados os procedimentos contidos no *Workflow*, como por exemplo: a criação de casos de testes baseados no critério todos *t-usos* estudado neste trabalho, a geração de elementos requeridos, entre outros.

Com base no critério todos *t-usos* proposto por Spoto, *et al.* (2005), no *Workflow* de teste do RUP e na Norma IEEE 829-1998, será descrita a construção do *Workflow* de teste voltado para ABDR através dos seguintes passos: plano de teste, projeção de teste, execução de teste e avaliação dos testes.

É importante ressaltar que, embora qualquer um dos critérios de teste estrutural possa ser aplicado no teste de unidade, esses critérios não contemplam o teste de *variáveis persistentes*, sendo que o tratamento de *variáveis persistentes* é realizado através dos critérios todos *t-usos* e todos os *dtu-caminhos,* proposto por Spoto (2000).

4.1 – Plano de Teste

Embora o custo e a complexidade do teste de um sistema sejam muito elevados, o planejamento de uma metodologia para testes bem definida e o uso de ferramentas adequadas podem aumentar a produtividade e efetividade dos testes (MASSONI, 1999).

Isso significa que se devem projetar testes, ou seja, elaborar planos de testes eficientes, tal que verifiquem: interação entre objetos e componentes implementados, se os requisitos foram corretamente implementados e se existe defeitos antes da implantação do *Software*.

No plano de teste são estabelecidos os requisitos a serem testados ou verificados, onde são gerados os elementos requeridos pelos critérios de teste estrutural de integração *Intra-classe* e *Inter-classe* para todas as variáveis tabelas, especificamente neste trabalho, os elementos requeridos são baseados no critério todos *t-usos*. São definidas também as 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 dependência *Intra-classe* e *Inter-classe* de Spoto (2000) de maneira automatizada. A Ferramenta JaBUTi, na versão atual vem sendo modificada para atender aos critérios de teste estrutural de unidade para ABDR e já identifica comandos de SQL para o teste de unidade (NARDI et.al., 2005).

Essencialmente, nesta fase os projetistas de testes também conhecidos como *Workers*, coletam e organizam informações sobre planejamento de testes, com essas informações os planos de testes são criados, resultando num relatório (artefato) que é o próprio plano de teste (MASSONI, 1999).

4.1.1 – Artefato Gerado no Plano de Teste

Os itens do *Software* a serem testados são identificados através do plano de teste, tais quais: o nível em que os itens devem ser testados, a abordagem utilizada para testar cada um dos itens, as tarefas envolvidas em cada atividade de teste, as pessoas responsáveis por cada atividade e os riscos associados ao plano de teste.

O plano de teste de *Software* é um documento onde se descreve o planejamento de todas as atividades envolvidas no teste de um *Software*. Um plano de teste deve conter também além das atividades, a extensão do teste, a abordagem utilizada no teste, os recursos necessários, o cronograma das atividades de teste e a definição do ambiente operacional para a execução dos testes (IEEE-Std-829, 1998).

Dependendo de fatores como a complexidade do produto em teste deve-se decidir se será elaborado um único plano de teste para as fases de teste de unidade, teste de integração, teste de sistema e teste de regressão, ou se para cada fase de teste será elaborado um plano de teste independente. Dessa forma, o plano de teste de *Software* pode ser um documento relacionado a um único projeto amplo de teste de *Software* ou pode ser um documento relacionado a um dos níveis de teste, tais como: plano de teste de unidade, plano de teste de integração, e plano de teste de sistema (IEEE-Std-829, 1998).

De acordo com a Norma IEEE 829-1998, as seções devem ser ordenadas na seqüência abaixo especificada. Caso seja necessário, as seções adicionais podem ser incluídas no final do documento, antes da seção referente às aprovações. Se todo o conteúdo ou parte de uma seção estiver em outro documento, então, poderá ser listada uma referência a esse material no lugar do conteúdo correspondente e esse material referenciado, deverá ser anexado ao plano de teste ou colocado à disposição dos usuários do plano de teste.

Será apresentada a seguir a estrutura que um plano de teste deve conter de acordo com a Norma IEEE 829-1998: identificador do plano de teste; introdução; itens de teste; funcionalidades e características do *Software* que devem e não devem ser testadas; abordagem; critérios de aprovação/reprovação de itens; critérios de suspensão e requisitos para a retomada do teste; produtos do teste; tarefas de teste; requisitos de ambiente, responsabilidades; equipe e treinamento necessários; cronograma; riscos e contingências; identificar as hipóteses e suposições; apêndices e aprovações.

4.2 – Projeção e Implementação de Teste

Nesta fase os projetistas de testes, procuram identificar um conjunto de casos de teste, identificam procedimentos de teste (que mostram como casos de teste são realizados), projetam funcionalidades específicas para os testes e geram *Scripts* de testes reutilizáveis. Os resultados desta fase são os modelos de teste, casos de teste, procedimentos de teste, classes, pacotes de projetos para teste e *Scripts* de testes (MASSONI, 1999).

Para que a qualidade esperada no processo de teste de *Software* seja alcançada, faz-se necessária a construção de casos de testes eficientes (com grande probabilidade de encontrar erros no sistema) para avaliar não somente a estrutura do sistema, mas também, as funcionalidades, bem como exercitar todos os caminhos lógicos e domínios de entrada e saída do sistema.

Além dos casos de testes, nesta fase são gerados também os grafos de programas, a instrumentação do código fonte e os elementos requeridos com base num determinado critério de teste.

4.2.1 – Instrumentação

A instrumentação de um programa original é feita na fase de projeção de testes e tem como objetivo obter informações sobre a execução de casos de teste, com intuito de analisar a cobertura dos elementos requeridos pelos diversos critérios de teste. De acordo com Spoto (2000), no caso de programas de aplicação, a instrumentação é dividida em duas etapas que se seguem:

- Inserção de comandos e informações no programa fonte, gerando assim uma nova versão do programa, usualmente denominada de unidade instrumentada para o teste de Unidade;
- A outra etapa de instrumentação visa capturar informações referentes à identificação das unidades em teste e sobre as tuplas envolvidas na execução dos comandos da SQL, para os testes *Intra-modular* e *Inter-modular*.

A consistência da instrumentação está essencialmente, na inserção de *pontas de prova* para indicar o número de cada bloco de comandos da linguagem hospedeira ou um comando executável da linguagem SQL, possibilitando desta forma, a identificação dos caminhos executados pelos casos de teste usados durante o teste. A *ponta de prova* é um comando de escrita do número e do nó num arquivo *Path*, produzindo a seqüência de execução dos nós em cada caso de teste durante o teste.

Um número identifica o módulo de programa e a unidade de programa no teste de integração, essa informação é escrita no início do arquivo *Path.tes*, representada pelo número *myyy* (*m* é o número do módulo e *yyy* é o número da unidade). A unidade UP_A do módulo $Mod_3.pc$ tem o número 3001 onde 3 representa o número do módulo e 001 representa o número da unidade no módulo $Mod_3.pc$ (SPOTO, 2000).

Uma instrumentação especial é utilizada para os critérios que tratam das associações *definição-t-uso*, que podem ser usadas tanto no teste de unidade como no teste de integração (*Intra-modular* e *Inter-modular*). O objetivo é informar as tuplas utilizadas durante a execução dos comandos de manipulação da SQL.

Essas informações são gravadas em um arquivo denominado *Keyint.tes* indicando o número de cada caso de teste e as informações sobre as tuplas utilizadas na execução do respectivo caso de teste. Esta instrumentação difere da primeira porque sua informação depende da composição dos campos que compõem a tupla (SPOTO, 2000).

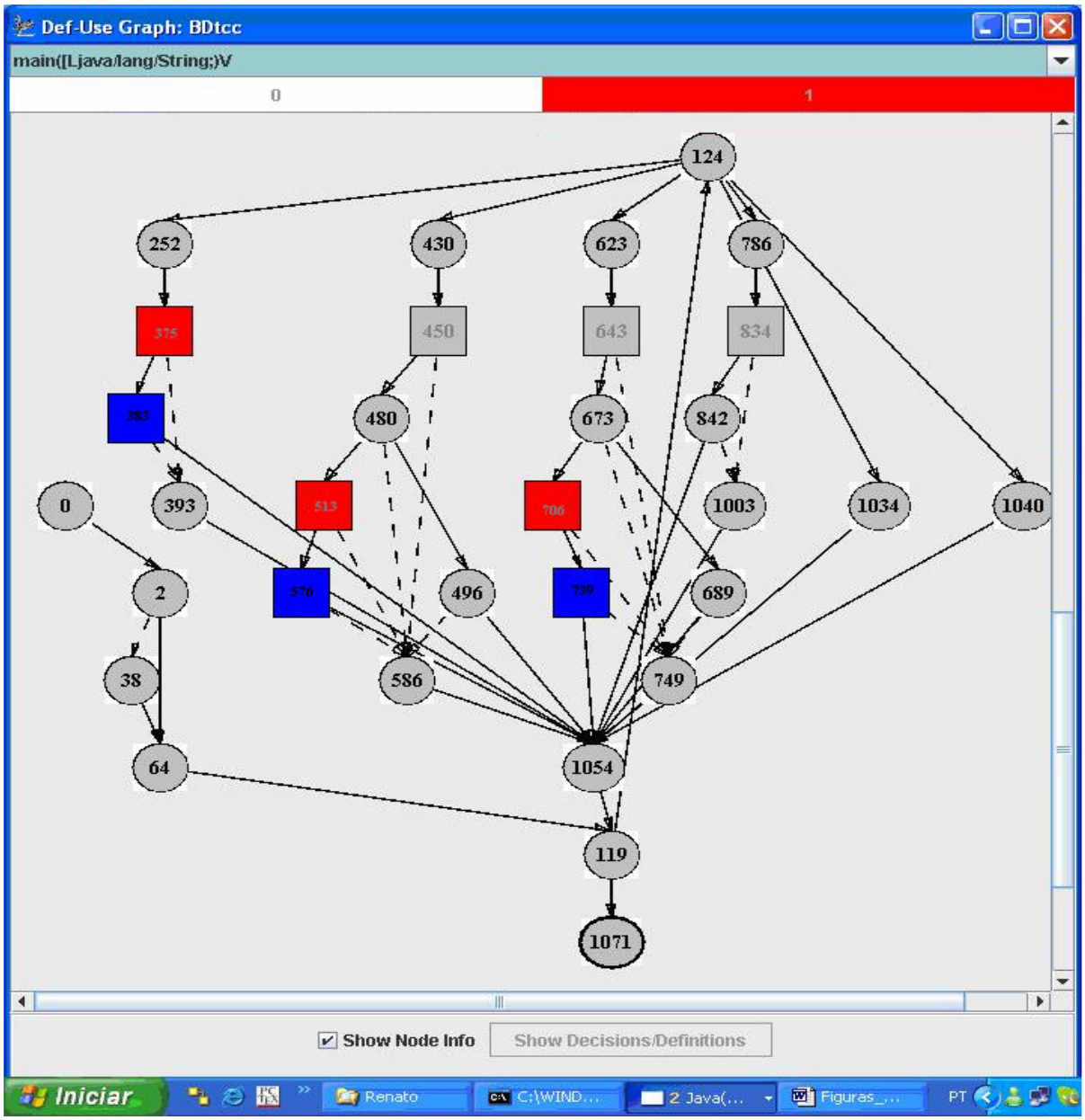

Figura 4.1 – Grafo de Programa do Estudo de Caso.

Para elaboração deste trabalho foi desenvolvido o código – para estudo de caso – demonstrado no (APÊNDICE A), que exercita os comandos executáveis SQL necessários para descrever o critério todos *t-usos (associações definições persistentes-uso*) proposto por Spoto (2000). O sistema se resume exclusivamente em manipular um BD através de comandos SQL (SELECT, INSERT, UPDATE E DELETE).

O desenvolvimento foi feito através da linguagem Java, e o BD foi criado em Firebird 1.5.3. Baseado no código mostrado no (APÊNDICE A), foi desenvolvido com o auxílio da Ferramenta de teste de *Software* JaBUTi v.1.0, um grafo de programa que está representado na Figura 4.1.

É apresentado no (APÊNDICE B) alguns trechos do código supramencionado instrumentado através da Ferramenta de teste de *Software* JaBUTi v.1.0.

4.2.2 – Geração dos Elementos Requeridos

A geração dos elementos requeridos, que é demonstrada na Figura 4.2 (BECARI, 2005), dar-se-á através de uma adaptação da definição de Spoto (2000), apresentada na seguinte notação: **{t,x,<i,j>,y,(k,m)}**, onde:

- **t:** é a *variável tabela*;
- **x:** é a identificação da classe e método da definição da *variável tabela*;
- **i:** é o nó de definição de t, onde ocorrem os comandos executáveis SQL (INSERT ou UPDATE ou DELETE);
- **j:** nó que ocorre a persistência da definição através do comando COMMIT, ou seja, torna a *definição persistente*;
- **y:** é a identificação da classe e método que ocorre o uso da mesma *variável tabela* (*t-uso*);

- **k:** nó que ocorre o uso da *variável tabela*, ou seja, o arco de *t-uso* onde os comandos SQL ocorrem e;
- **m:** é o arco de saída de um comando DML.

Esse critério tem como objetivo requerer que todo sub-caminho que inicia no nó i da SQL (INSERT ou DELETE ou UPDATE) e passa pelo nó j (COMMIT), onde ocorre a *definição persistente* de ***t***, e alcança um arco de saída do nó da SQL, onde ocorre o *uso* de ***t*** (*t-uso*) com os comandos (INSERT, DELETE, UPDATE ou SELECT), sejam executados pelo menos uma vez para a mesma tupla (SPOTO, 2000).

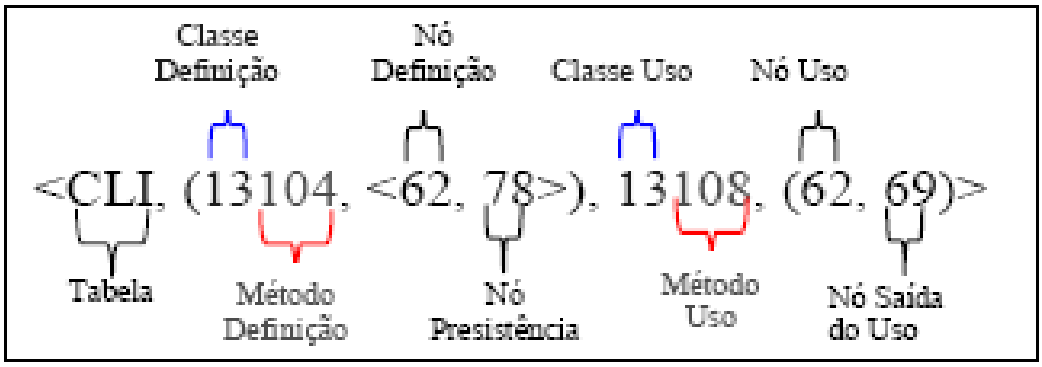

Figura 4.2 – Identificação dos Elementos Requeridos.

Com base na Figura 4.2 (BECARI, 2005), tem-se:

- **CLI**: representa a *variável tabela*;
- **13104**: indica que está na classe 13000, no método 104 da 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 e;
- **69**: nó de saída do uso da *variável tabela* CLI.

Serão apresentados a seguir, os elementos requeridos (associações) referente ao critério todos *t-usos*, que foram gerados com base no estudo de caso deste trabalho, para este exemplo foi adotado o número concatenado pelos números: 10 que representa a classe e o número 100 que representa o método:

1) {PROFESSOR, 10100, <375,383>, 10100, (834,842)} - <Insert/Select>
2) {PROFESSOR, 10100, <375,383>, 10100, (834,1003)} - <Insert/Falha no Select>
3) {PROFESSOR, 10100, <375,383>, 10100, (375,383)} - <Insert/Insert>
4) {PROFESSOR, 10100, <375,383>, 10100, (375,393)} - <Insert/Falha no Insert>
5) {PROFESSOR, 10100, <375,383>, 10100, (450,480)} - <Insert/Select Update>
6) {PROFESSOR, 10100, <375,383>, 10100, (450,586)} - <Insert/Falha no Select Update>
7) {PROFESSOR, 10100, <375,383>, 10100, (643,673)} - <Insert/Select Delete>
8) {PROFESSOR, 10100, <375,383>, 10100, (643,749)} - <Insert/Falha no Select Delete>

9) {PROFESSOR, 10100, <513,576>, 10100, (834,842)} - <Update/Select>
10) {PROFESSOR, 10100, <513,576>, 10100, (834,1003)} - <Update/Falha no Select>
11) {PROFESSOR, 10100, <513,576>, 10100, (375,383)} - <Update/Insert>
12) {PROFESSOR, 10100, <513,576>, 10100, (375,393)} - <Update/Falha no Insert>
13) {PROFESSOR, 10100, <513,576>, 10100, (450,480)} - <Update/Select Update>
14) {PROFESSOR, 10100, <513,576>, 10100, (450,586)}-<Update/Falha no Select Update>
15) {PROFESSOR, 10100, <513,576>, 10100, (643,673)} - <Update/Select Delete>
16) {PROFESSOR, 10100, <513,576>, 10100, (643,749)} - <Update/Falha no Select Delete>

17) {PROFESSOR, 10100, <706,739>, 10100, (834,842)} - <Delete/Select>
18) {PROFESSOR, 10100, <706,739>, 10100, (834,1003)} - <Delete/Falha no Select>
19) {PROFESSOR, 10100, <706,739>, 10100, (375,383)} - <Delete/Insert>
20) {PROFESSOR, 10100, <706,739>, 10100, (375,393)} - <Delete/Falha no Insert>
21) {PROFESSOR, 10100, <706,739>, 10100, (450,480)} - <Delete/Select Update>
22) {PROFESSOR, 10100, <706,739>, 10100, (450,586)} - <Delete/Falha no Select Update>
23) {PROFESSOR, 10100, <706,739>, 10100, (643,673)} - <Delete/Select Delete>
24) {PROFESSOR, 10100, <706,739>, 10100, (643,749)} - <Delete/Falha no Select Delete>

4.2.3 – Geração dos Casos de Testes

Os critérios de teste propostos 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) (SPOTO, 2000).

A partir dos elementos requeridos que foram apresentados na Seção anterior, segue alguns exemplos de casos de testes que foram gerados com intuito de exercitar alguns desses elementos requeridos, consideram-se para os caminhos que serão apresentados, que não ocorrerá erro de conexão com o BD:

1 - Inserindo uma tupla e consultando a mesma tupla inserida.
Dados de Teste: 10, Edmundo, Marilia, 34026969;
Caso Teste: #1;
Caminho percorrido: 10100 0 2 64 119 124 252 375 383 1054 119 124 786 834 842 1054 119;
Tupla usada: PROFESSOR = 10;
Elemento requerido satisfeito: 1.

2 - Alterando uma tupla e consultando a mesma tupla alterada.
Dados de Teste: 20, 34027080;
Caso Teste: #2;
Caminho percorrido: 10100 0 2 64 119 124 430 450 480 513 576 1054 119 124 786 834 842 1054 119;
Tupla usada: PROFESSOR = 20;
Elemento requerido satisfeito: 9.

3 - Excluindo uma tupla e seguidamente tentando consultar a mesma tupla excluída.
Dados de Teste: 30;
Caso Teste: #3;
Caminho percorrido: 10100 0 2 64 119 124 623 643 673 706 739 1054 119 124 786 834 1003 1054 119;
Tupla usada: PROFESSOR = 30;
Elemento requerido satisfeito: 18.

4.2.4 – Artefatos Gerados na Projeção e Implementação de Teste

Nesta seção serão apresentados os artefatos que poderão ser gerados neste segundo passo referente ao desenvolvimento do *Workflow* de teste. Segundo a Norma IEEE-std-829-1998 se houver a necessidade de se gerar outro artefato, como por exemplo, a especificação de projeto de teste, formulários adicionais podem ser utilizados sendo que estes devem ser devidamente anexados.

De acordo com a Norma IEEE-std-829-1998, as seções devem ser ordenadas na seqüência especificada em cada um dos artefatos gerados nesta fase. Assim como no plano de teste, as seções adicionais poderão ser incluídas no final. Caso todo conteúdo ou parte de uma seção estiver em outro documento, deve-se fazer a devida referência ao documento, e este material referenciado deverá estar anexado ao respectivo artefato gerado, ou ser colocado à disposição dos usuários do artefato em questão.

4.2.4.1 – Especificação de Caso de Teste

Este documento define um caso de teste identificado por uma especificação de projeto de teste, que é uma versão refinada do plano de teste. A Norma justifica a separação deste documento em relação ao documento especificação de projeto de teste tanto para

permitir seu uso em mais de um projeto de teste quanto para sua reutilização nos testes de outros produtos de *Software*. Por exemplo, a especificação do caso de teste de uma rotina de validação de CPF pode ser utilizada para o teste de todos os produtos que possuam esta funcionalidade.

Visto que um caso de teste pode ser referenciado por várias especificações de projeto de teste, que por sua vez pode ser utilizado durante um longo período de tempo por diferentes grupos de teste, as informações específicas devem estar incluídas na especificação de caso de teste para permitir o reuso.

Será apresentada a seguir a estrutura que uma Especificação de caso de teste deve conter de acordo com a Norma IEEE 829-1998: identificador da especificação de caso de teste; itens de testes; especificações de entrada; especificações de saída; requisitos de ambiente de teste; requisitos de procedimentos especiais; dependências entre casos de Teste.

4.2.4.2 – Especificação de Procedimento de Teste

Este documento especifica os passos necessários para executar um conjunto de casos de teste ou, de um modo geral são os passos usados para analisar um item de *Software* com o objetivo de avaliar um conjunto de funcionalidades e características.

A Norma não especifica como agrupar os casos de teste, caso seja necessário devem-se analisar os casos de teste que possuem relacionamentos ou dependências, ou que possuam os mesmos requisitos de ambiente. A Norma justifica ainda a separação deste documento em relação ao documento de especificação de caso de teste, pois um procedimento de teste deve possuir somente passos simples, não devendo conter outros detalhes.

Será apresentada a seguir a estrutura que uma especificação de procedimento de teste deve conter de acordo com a Norma IEEE 829-1998: identificação da especificação de

procedimento de teste; propósito do procedimento de teste; requisitos especiais; passos do procedimento de teste.

4.3 – Execução de Teste

Nesta fase os executores de testes (integração, sistema e desempenho), executam testes, revisam resultados e registram defeitos, tendo como resultando os defeitos encontrados (MASSONI, 1999).

Durante a execução dos casos de testes ou execução do teste propriamente dito, são gerados dois arquivos: *Pathint.tes* e o *Keyint.tes*, cujo conteúdo será descrito a seguir (SPOTO, 2000):

- *Pathint.tes:* armazena o número do caso de teste, mais o número da unidade exercitada e o respectivo caminho executado;
- *Keyint.tes:* armazena o número de cada caso de teste, o número da unidade executada e a tupla utilizada para exercitar o caso de teste (em alguns casos pode ser a própria chave primária).

Com base no caso de teste #1 (insere uma tupla e consulta a mesma tupla inserida na variável *tabela* PROFESSOR) referente aos casos de testes gerados no estudo de caso deste trabalho, segue um exemplo de cada um dos arquivos supramencionados:

- *Path* : #1 10100 0 2 64 119 124 252 375 383 1054 119 124 786 834 842 1054 119;
- *Key* : #1 (10100 NUM_PROF=10);

Através da execução dos casos de teste, obtém-se posteriormente a análise de cobertura para avaliação do grau de satisfação de um critério aplicado num teste estrutural. A análise de cobertura é feita através da instrumentação de cada unidade do programa em teste,

pela monitoração dos caminhos executados pelo conjunto de casos de teste e pela determinação dos elementos requeridos executados e não executados (SPOTO, 2000).

Segundo Becari (2005), 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. 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).

4.3.1 – Artefato Gerado na Execução de Teste

Como foi apresentado no Capítulo 2 que trata da Norma IEEE-std-829-1998, este documento denominado diário de teste, que pode ser gerado nesta fase, apresenta registros cronológicos dos detalhes relevantes relacionados com a execução dos testes.

Segundo a Norma IEEE-std-829-1998, se houver a necessidade de se gerar outro artefato referente a esta fase, como por exemplo, o relatório de incidente de teste, formulários adicionais podem ser utilizados, sendo que estes devem ser devidamente anexados.

De acordo com a Norma IEEE-std-829-1998, as seções devem ser ordenadas na seqüência especificada em cada um dos artefatos gerados nesta fase. Assim como no plano de teste, as seções adicionais poderão ser incluídas no final. Caso todo conteúdo ou parte de uma seção estiver em outro documento, deve-se fazer a devida referência ao documento, e este material referenciado deverá estar anexado ao respectivo artefato gerado, ou ser colocado à disposição dos usuários do Artefato em questão.

Será apresentada a seguir a estrutura que um diário de teste deve conter de acordo com a Norma IEEE 829-1998: identificador do diário de teste; descrição do teste e registros de atividades e eventos.

4.4 – Avaliação dos Testes

Nesta fase analisa-se a cobertura que os casos de testes gerados tiveram em relação a um determinado critério.

Após a execução de todos os casos de teste, é feita de forma manual a avaliação da cobertura do critério, avaliando cada caso de teste, tuplas e quais unidades foram envolvidas, para depois avaliar quais associações foram satisfeitas. Essa avaliação pode ser feita utilizando um autômato finito que avalie o $grafo_d$<l,*i*> que inicia na definição persistente e no $grafo_u$<l,*i*> que inicia no nó de entrada do grafo de programa (da unidade associada) e termina nos nós que contêm um *t-uso* (SPOTO, 2000).

Os projetistas de testes estabelecem métricas do progresso dos testes e geram relatórios de avaliação. O resultado desta fase é o relatório de avaliação dos testes ou relatório – resumo de teste (MASSONI, 1999).

Com relação à cobertura de teste do estudo de caso deste trabalho, podemos observar que os casos de teste 1, 2 e 3 cobrem as associações supramencionadas de números: 1, 9 e 18. Os números dos casos de teste anotados na frente das associações representam as seqüências de casos de testes necessárias para cobrir a associação. Neste exemplo foram apresentados apenas alguns casos de teste com intuito de ilustrar a cobertura dos elementos requeridos (*associações def-t-uso de ciclo1*).

4.4.1 – Artefato Gerado na Avaliação dos Testes

Após as atividades de teste pode-se gerar o relatório – resumo de teste, como foi apresentado no Capítulo 2 que trata da Norma IEEE-std-829-1998, este documento apresenta

de forma resumida os resultados das atividades de teste associadas com uma ou mais especificação de projeto de teste e provê avaliações baseadas nesses resultados.

Segundo a Norma IEEE-std-829-1998, as seções devem ser ordenadas na seqüência especificada em cada um dos artefatos gerados nesta fase. Assim como nos artefatos apresentados anteriormente, as seções adicionais poderão ser incluídas no final. Caso todo conteúdo ou parte de uma seção estiver em outro documento, deve-se fazer a devida referência ao documento, e este material referenciado deverá estar anexado ao respectivo artefato gerado, ou ser colocado à disposição dos usuários do artefato em questão.

Será apresentada a seguir a estrutura que um relatório – resumo de teste deve conter de acordo com a Norma IEEE 829-1998: identificador do relatório-resumo de teste; resumo dos itens de teste; desvios das especificações; avaliação de abrangência do teste; resumo de resultados; avaliação dos itens de teste; resumo de atividades e aprovações.

4.5 – Execução do *Workflow* de Teste de ABDR na FADAT

De acordo com a Ferramenta FADAT, que foi apresentada no Capítulo 3, Seção 4.2 deste trabalho, segue algumas telas que representam o processo de utilização desta ferramenta de apoio à documentação de aplicações de teste de *Software.*

As telas demonstradas a seguir, exemplificam a inserção de informações relevantes para a documentação de um *Workflow* de teste de *Software* voltado para ABDR.

É importante salientar que as telas que serão apresentadas referem-se somente a atividade de plano de teste do *Workflow*, de tal forma que, caso seja necessário documentar outros tipos de fases de teste referente aos passos do *Workflow* de teste, deve-se realizar um procedimento semelhante para os outros artefatos, tendo em vista que a FADAT segue um modelo padronizado.

Os arquivos gerados que representam os artefatos de saída da Ferramenta FADAT, contém a estrutura prevista na Norma IEEE-std-829-1998.

Figura 4.3 – Tela de Apresentação da FADAT.

Figura 4.4 – Tela de Cadastro de Usuário da FADAT.

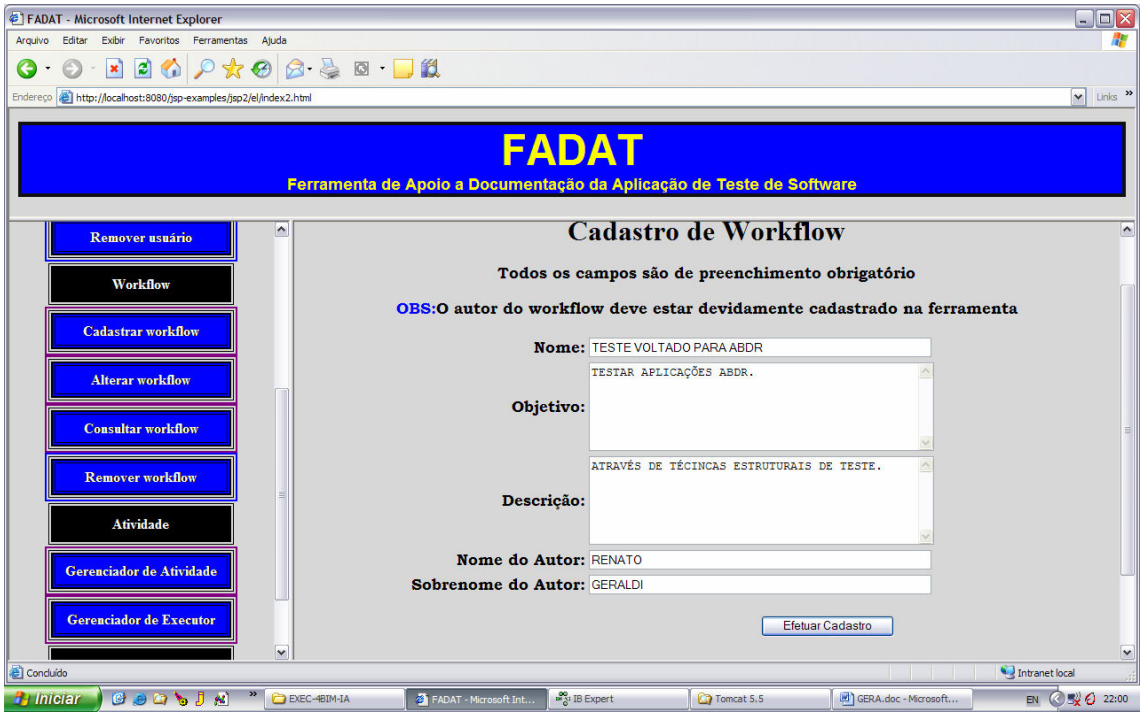

Figura 4.5 – Tela de Cadastro de *Workflow* da FADAT.

Figura 4.6 – Tela de Atribuição de Tarefa ao Executor de um determinado *Workflow*.

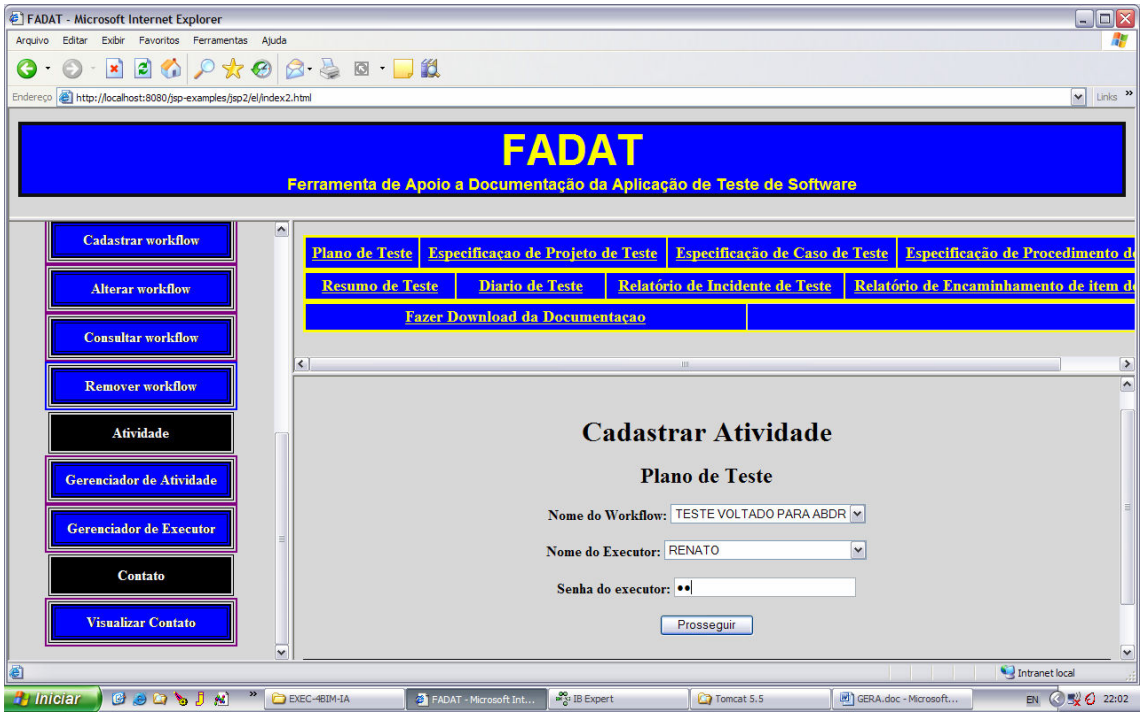

Figura 4.7 – Tela de Cadastro de Atividade da FADAT.

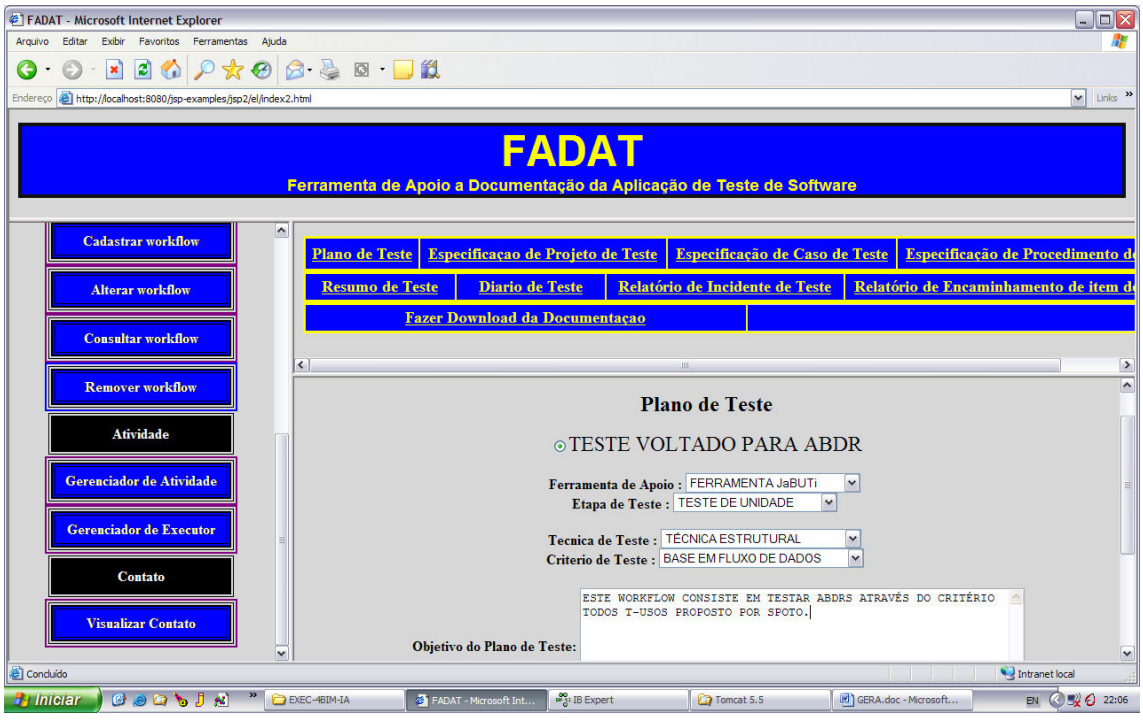

Figura 4.8 – Tela Responsável pela Entrada de todas as Informações Necessárias referente à Documentação da Atividade de Plano de Teste.

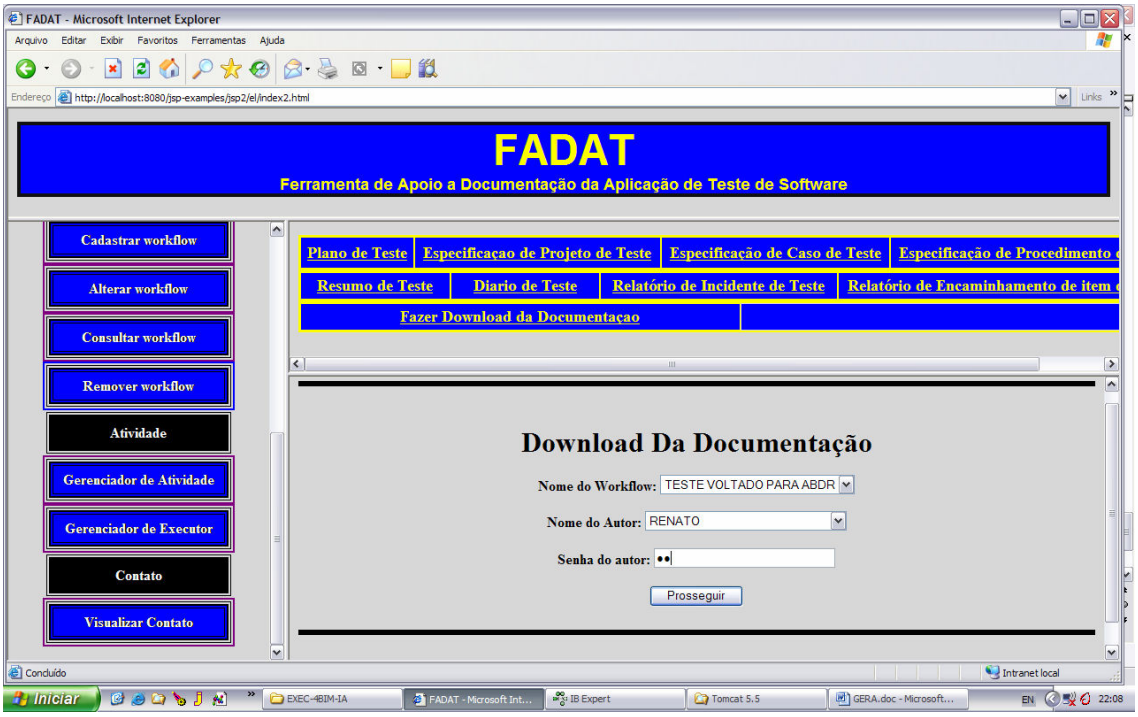

Figura 4.9 – Tela de *Login* para *Download* de Artefatos de um determinado *Workflow*.

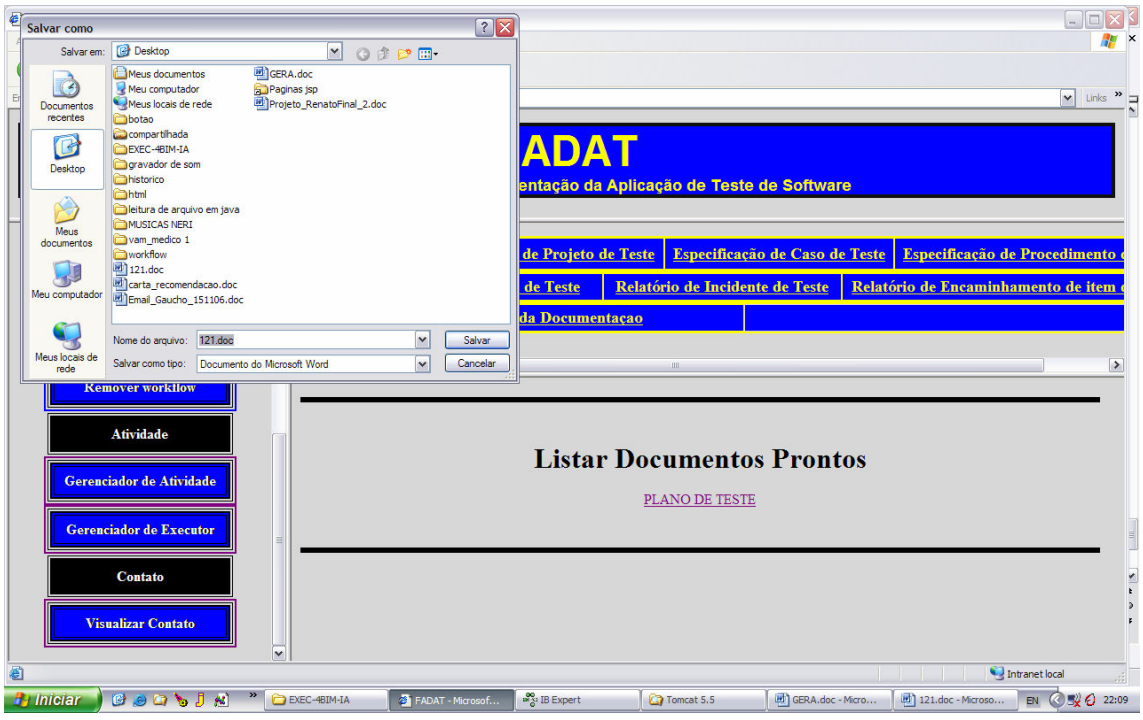

Figura 4.10 – Tela para *Download* de Artefatos referentes às Atividades relacionadas a um determinado *Workflow*.

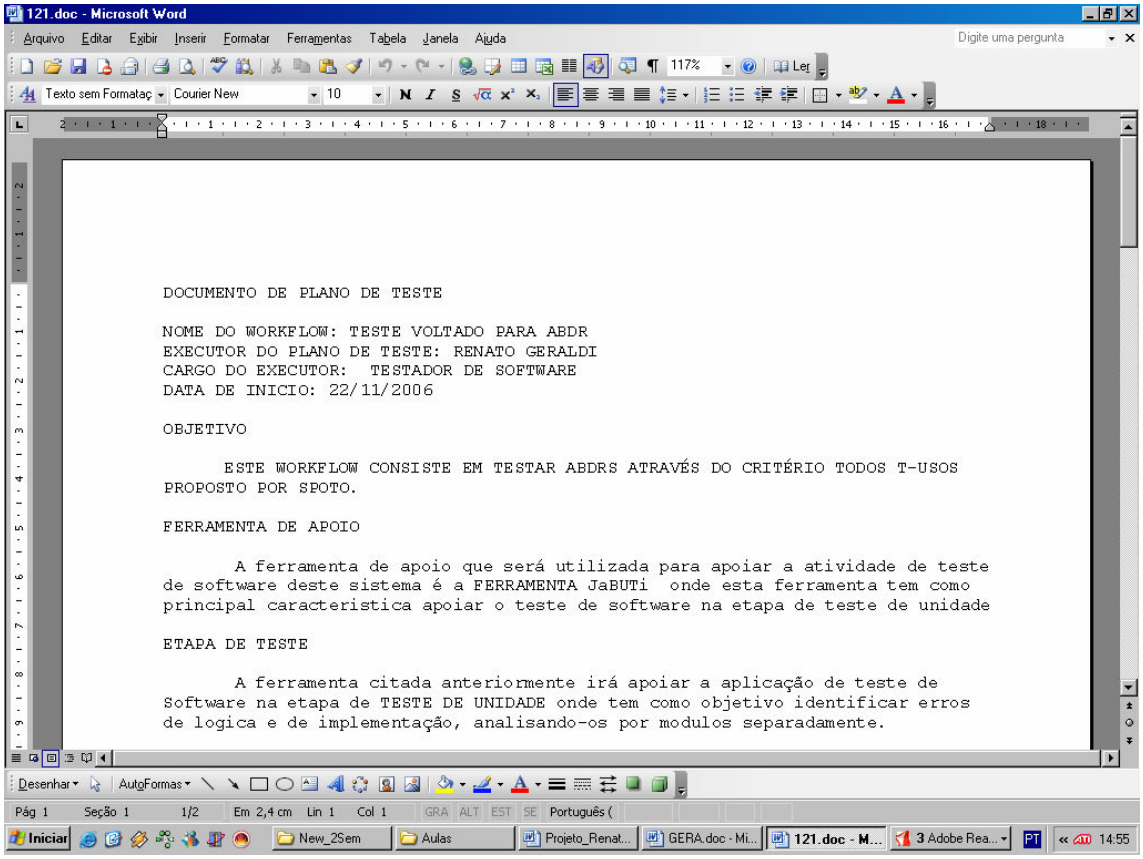

Figura 4.11 – Arquivo Gerado pela FADAT correspondente ao Artefato de Saída, referente à Atividade de Plano de Teste.

CONCLUSÕES

O principal objetivo deste trabalho, que é a criação de um *Workflow* de teste voltado às ABDRs, foi atendido visando assim dar uma estruturação e organização melhorada nos procedimentos que envolvem o teste de *Software* de ABDR*,* através de uma seqüência de passos pré-definidos. É possível observar que um *Workflow* de teste, auxilia e muito aos engenheiros de *Software* e a todos os profissionais envolvidos no teste de *Software*, principalmente no que diz respeito ao gerenciamento do teste.

Através da utilização de um *Workflow* de teste, o produto de *Software* resultante da fase de teste, obterá uma maior qualidade de *Software,* reduzindo assim o tempo e o custo da aplicação de teste de *Software* em ABDRs, contribuindo de forma positiva com o resultado do produto final do desenvolvimento de um *Software*, que é o sistema pronto disponível para o Usuário.

É importante salientar que o *Workflow* gerado neste trabalho pode-se tornar ainda mais eficiente aos usuários de teste de *Software,* a partir da integração usando uma ferramenta de apoio que gerencia e documenta todo processo de teste. No caso do *Workflow* proposto neste trabalho, essa integração utilizou-se a Ferramenta FADAT desenvolvida por Parckert (2006), como um projeto acadêmico, que tem como principal objetivo apoiar à documentação necessária à aplicação de teste de *Software*.

Com base neste trabalho algumas perspectivas futuras podem ser analisadas, com intuito de tornar à aplicação deste *Workflow* de teste voltado para ABDR mais eficiente e eficaz. Podem ser inclusos mais critérios de testes de *Software* referentes à técnica estrutural, como por exemplo, o critério todos *dtu-caminhos* proposto por Spoto (2000), também aplicado em ABDRs.

REFERÊNCIAS BIBLIOGRÁFICAS

ACREE, A. T., *et al., Relatório Técnico GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Ga.*, Setembro, 1979.

ATLEE, Joanne, *Workflow Management Systems - A Standard Architecture for Automating Workflows*. Tutorial, Dezembro, 1997.

BATISTA, Denerval M., DBValTool: Uma Ferramenta para Apoiar o Teste e a Validação do Projeto do Banco de Dados Relacional, Dissertação de mestrado, UFPR, Curitiba, PR, 2003.

BECARI, Gisele M., A Eficácia de Critérios de Teste Estrutural em Aplicação de Banco de Dados Relacional. Dissertação de Mestrado, CPGCC, UNIVEM, SP, 2005.

BEIZER, B., *Black-Box Testing: Techniques for Funcional Testing of Software and System*; Wiley, *New York*, 1995.

BUDD, T. A., *et al.*, *"Theoretical and Empirical Studies on Using Prog Mutation to Test the Functional Correctness of Prog."*, 7th *ACM Symposium on Principles of Programming Languages*, Janeiro, 1980.

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

CRAIG, R. D.; JASKIEL, S. P., *Systematic Software Testing*., *Artech House Publishers*, 2002.

CRESPO, A. N.; MARTINEZ, Márcia R. M.; JINO M.; JÚNIOR, Miguel T. A., Documentação de Teste – Referência: Norma IEEE std 829 - 1998, CenPRA – Centro de Pesquisas Renato Archer, Campinas, São Paulo, Setembro, 2000.

CRESPO, A. N.; SILVA O. J.; BORGES C. A.; SALVIANO C. F.; JINO M., Uma Metodologia para Teste de *Software* no Contexto da Melhoria de Processo, Campinas, São Paulo, 2004.

DELAMARO, M. E., Proteum – Um Ambiente de Teste Baseado na Análise de Mutantes. Dissertação de Mestrado. ICMC/USP, São Carlos, Outubro, 1993.

DEMILLO, R. A., *Software Testing and Evaluation. The Benjamim/Commings Publishing Company*, Inc, 1978.

DORIA, E. S., Replicação de Estudos Empíricos em Engenharia de *Software*. Dissertação de Mestrado. ICMC/USP, São Carlos, Maio, 2001.

ELMASRI, R. and NAVATHE, S. B., "*Fundamentals of Database Systems*", Sec. Edition, Addison Wesley, 1994.

FRANKL, F. G.; WEYUKER, E. J., "*An Applicable Family of Data Flow Testing Criteria" IEEE Trans on Software Eng.*, Vol. 14, No. 10, pp.1483-1498, *October* 1988.

IEEE-Std-829, *IEEE*: *Standard for Software Test Documentation, Software Engineering Technical Committee of the IEEE Computer Society, September*, 1998.

ISO/IEC TR 15504-1, *Information technology – Software Process Assessment, Part 1: Concepts and introductory guide*, 1998.

KRUCHTEN, P. "*The Rational Unified Process an Introduction*", 2ª edição, editora Addison – Wesley, 2000.

MALDONADO, J. C.; VICENZI, Auri M. R.; BARBOSA E. F.; SOUZA S. R. S.; DELAMARO, M. E., Aspectos Teóricos e Empíricos de Teste de Cobertura de *Software*. Notas do ICMC, 31. Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Junho de 1998.

MASSONI, T. L., Estudo do RUP: Projetar Testes de *Software*, DI/UFPE, Pernambuco, 1999.

MATOS, E. S., Revisão Bibliográfica: *Workflow* para Testes de *Software*, Dissertação de Mestrado. UFCG/Campina Grande, PB, Novembro, 2004.

MORO, Mirella. M., "*Workflow* na *WEB"*, Trabalho final entregue para disciplina de Sistema de Banco de dados Distribuídos. Instituto de Informática da UFRGS, RS, 1998. Disponível em : http://www.inf.ufrgs.br/~mirella/*workflow*/home.html, acesso em Maio de 2006.

MYERS, G., "*The Art of Software Testing*", Wiley, New York, 1979.

NARDI, Paulo A.; DELAMARO, M. E.; SPOTO, E. S.; VINCENZI, A. M. R., "JaBUTi/BD: Utilização de Critérios Estruturais em Aplicações de Banco de Dados Java, 20º Simpósio Brasileiro de Qualidade de *Software* – SBES; Uberlândia – MG. 12ª Sessão de Ferramentas – Outubro, 2005.

NARDI, Paulo A., "Inclusão do Critério Todos-T-Usos na Ferramenta JaBUTi", Dissertação de Mestrado, UNIVEM, Marilia, SP, Março, 2006.

PARCKERT, Jeferson D., "Aspectos de automatização da documentação da aplicação de *Workflow* de Teste.", Trabalho de Conclusão de Curso de Graduação, UNIVEM, Marília, SP, Novembro, 2006.

PHILLIPS, Dwayne, *The Software Project Manager's Handbook, IEEE Computer Society*, 1998.

PRESSMAN, R. S., *Software Engineering: A practitioner's approach*. Quinta edição. McGraw-Hill, 2000.

PRESSMAN, R. S., *Software engineering: A practitioner's approach*. Sexta edição. McGraw-Hill, 2005.

SOMMERVILLE, I., Engenharia de *Software*. Sexta edição. Pearson Addilson Wesley, 2003.

SPOTO, Edmundo S., Um Estudo de Critérios de Teste de *Software* Baseados em Fluxo de Dados. Campinas: UNICAMP-FEE-DCA, Tese de Graduação, 1995.

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

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 - Maio, 2005.

VINCENZI, Auri M.R.; MALDONADO, José C.; DELAMARO, Marcio E.; SPOTO, Edmundo S.; WONG, Eric, Capítulo 8 *Software* Baseado em Componentes: Uma Revisão sobre Teste. Springer, 2003.

WHITE, Lee J., "*Software Testing and Verification*", *Advances in Computers* – Vol.26, Ed. Acad. Press, Inc., 1987.

ZALLAR, K., *Are You Ready for Test Automation Game ?, STQE – Software Testing and Quality Engineering Magazine*, Vol.3, No. 6, Nov/Dec 2001.

APÊNDICE A

Nesta seção segue o código utilizado no *Workflow* deste projeto, referente ao estudo de caso deste projeto:

```
import java.util.*;   import java.io.*;   import java.awt.*;   import java.lang.*;   import java.sql.*;

public class BDtcc
{   public static void main(String[] args)throws Exception
    {   Connection conexao =  null;
        try // CONECTANDO BD
        {   String url = "jdbc:firebirdsql:127.0.0.1/3050:c:/TCC/BDTCC.gdb";
            Class.forName("org.firebirdsql.jdbc.FBDriver");
            conexao = DriverManager.getConnection(url,"SYSDBA","masterkey");
            System.out.println("CONEXAO COM O BD FIREBIRD 1.5 REALIZADA COM SUCESSO !!!");
            conexao.setAutoCommit(false);
        }catch (SQLException e)
        {   System.out.println("ERRO NA CONEXAO COM O BD FIREBIRD 1.5 : "+ e); }

        BufferedReader key = new BufferedReader(new InputStreamReader(System.in));
        Statement st = conexao.createStatement();
        ResultSet rs = null;

        String sql = null;      String num = null;      boolean sai = false;    String nome = null;     String fone = null;
        String aux = null;      String cidade = null;   boolean reg = false;    int opc = 0;

        while (!sai)
        {   System.out.println();
            System.out.println("########     MENU PRINCIPAL     ########");
            System.out.println("       1- INSERIR");
            System.out.println("       2- ALTERAR");
            System.out.println("       3- DELETAR");
            System.out.println("       4- CONSULTAR");
            System.out.println("       0- SAIR");
            System.out.print("ESCOLHA UMA OPCAO : ");

            aux = key.readLine();
            opc = Integer.parseInt(aux);
            sql = num = nome = cidade = fone = null;

            switch (opc)
            {   case 1 : // INSERE
                System.out.println();
                System.out.print("DIGITE O NUMERO DO PROF...: ");
                num = key.readLine();
                System.out.print("DIGITE O NOME DO PROF.....: ");
                nome = key.readLine();
                System.out.print("DIGITE A CIDADE DO PROF...: ");
                cidade = key.readLine();
                System.out.print("DIGITE O FONE DO PROF.....: ");
                fone = key.readLine();
                sql = "INSERT INTO PROFESSOR VALUES (";
                sql = sql + num + ",'"+nome+"','"+cidade+"','"+fone+"')";
```

```
 try
 {   st.executeUpdate(sql);
     conexao.commit();
 }catch(SQLException e)
 {   System.out.println();
     System.out.println("ERRO NO COMANDO INSERT : " + e); }
 break;

 case 2 : // ALTERA
 System.out.println(); // LÊ DO USUÁRIO O NUMERO DO PROFESSOR
 System.out.print("DIGITE O NUMERO DO PROFESSOR A SER ALTERADO : ");
 num = key.readLine();
 try
 {   sql = "SELECT * FROM PROFESSOR WHERE NUM_PROF = "+num;
     rs = st.executeQuery(sql);
     reg = rs.next();
     if (!reg)
     {   System.out.println();
         System.out.println("PROFESSOR NAO CADASTRADO !!!");
     }else
     {   System.out.println(); // LÊ O NOVO FONE DO PROFESSOR
         System.out.print("DIGITE O NOVO FONE : ");
         fone = key.readLine();
         sql = null;
         sql = "UPDATE PROFESSOR SET FONE = '"+fone+"' WHERE NUM_PROF = "+num;
         st.executeUpdate(sql);
         conexao.commit(); } // FIM DO ELSE
 }catch(SQLException e)
 {   System.out.println();
     System.out.println("ERRO NO COMANDO UPDATE : " + e); }
 break;

 case 3 : // DELETA
 System.out.println(); // LÊ DO USUÁRIO O NUMERO DO PROFESSOR
 System.out.print("DIGITE O NUMERO DO PROFESSOR A SER DELETADO : ");
 num = key.readLine();
 try
 {   sql = "SELECT * FROM PROFESSOR WHERE NUM_PROF = "+num;
     rs = st.executeQuery(sql);
     reg = rs.next();
     if (!reg)
     {   System.out.println();
         System.out.println("PROFESSOR NAO CADASTRADO !!!");
     }else
     {   sql = null;
         sql = "DELETE FROM PROFESSOR WHERE NUM_PROF = "+num;
         st.executeUpdate(sql);
         conexao.commit(); } // FIM DO ELSE
 }catch(SQLException e)
 {   System.out.println();
     System.out.println("ERRO NO COMANDO DELETE : " + e); }
 break;

 case 4 : // CONSULTA
 System.out.println();
 System.out.print("DIGITE O NUMERO DO PROF.: ");
 num = key.readLine();
 System.out.println();
 sql = "SELECT * FROM PROFESSOR WHERE NUM_PROF = "+num;
```

```
                try
                 {   rs = st.executeQuery(sql);
                     rs.next();
                     num = rs.getString("NUM_PROF");  System.out.println("NUMERO...: " + num);
                     nome = rs.getString("NOME");     System.out.println("NOME.....: " + nome);
                     cidade = rs.getString("CIDADE"); System.out.println("CIDADE...: " + cidade);
                     fone = rs.getString("FONE");     System.out.println("FONE.....: " + fone);
                 }catch(SQLException e)
                 {   System.out.println("ERRO NO COMANDO SELECT : " + e); }
                 break;

                 case 0 : // ENCERRA EXECUCAO
                 sai = true;
                 break;

                 default :
                 System.out.println();
                 System.out.println("OPCAO INVALIDA !!!");
               } // FIM DO SWITCH
               System.out.println();
               System.out.println("ITERACAO REALIZADA...");
            } // FIM DO WHILE
            rs.close();
            st.close();
            conexao.close(); // FECHA A CONEXÃO
        } // FIM DA CLASS Main
    } // FIM DA CLASS Banco
```

APÊNDICE B

Nesta seção segue alguns trechos do código utilizado no *Workflow,* apresentado no (APÊNDICE A), referente ao estudo de caso deste projeto, que foi instrumentado através da Ferramenta de Teste de *Software* JaBUTi v.1.0.

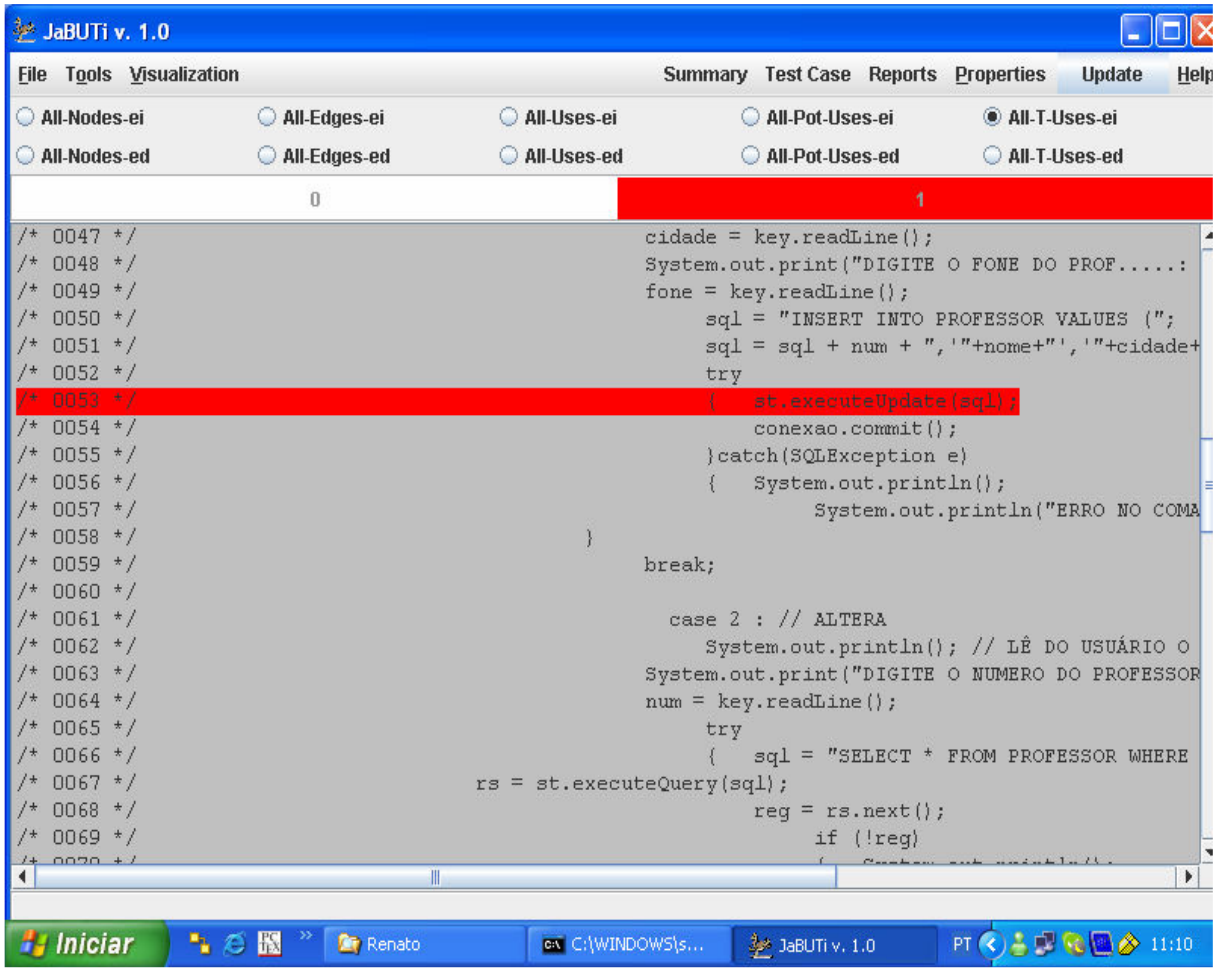

Figura B.1 – Trecho de Código de Programa referente ao Estudo de Caso – Parte 1.

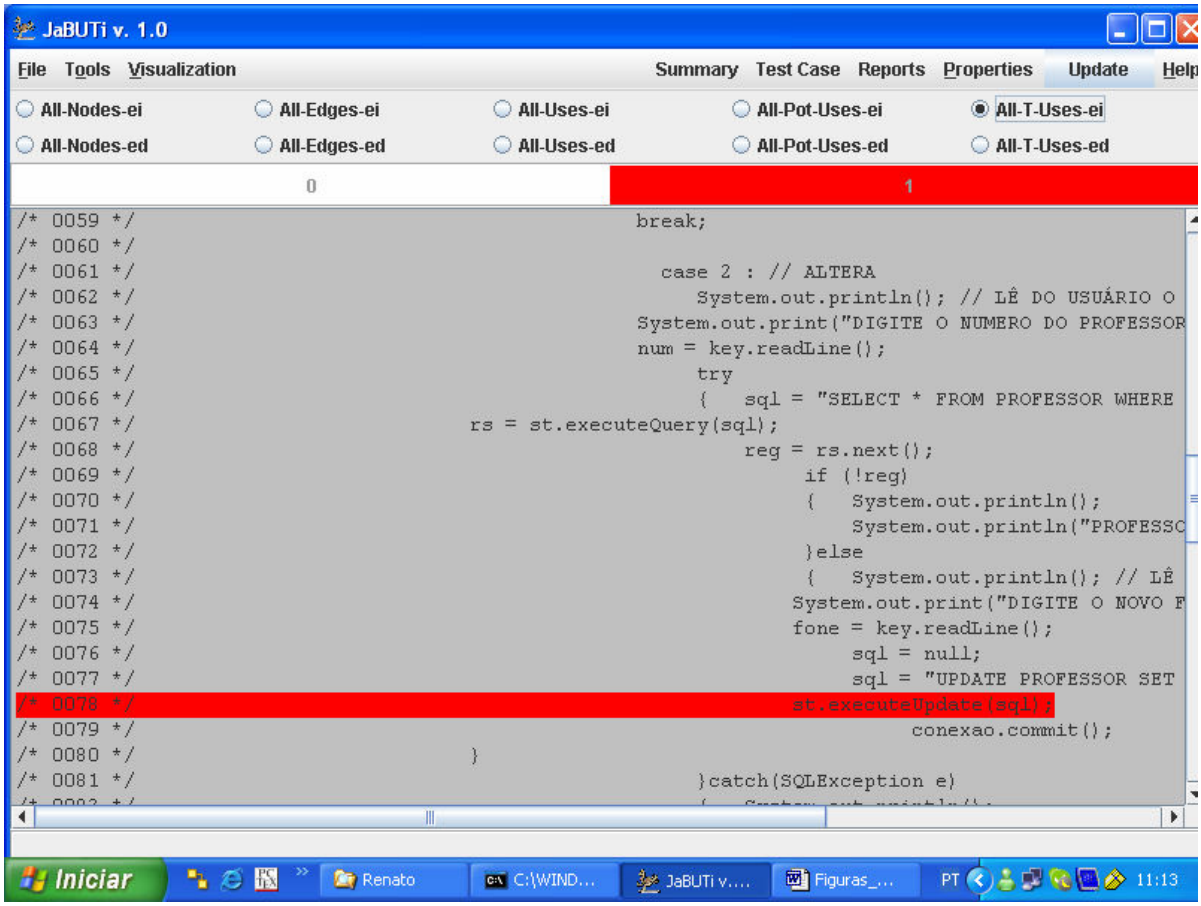

Figura B.2 – Trecho de Código de Programa referente ao Estudo de Caso – Parte 2.