FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

NÍVIA TOGNI CÊGA DOS SANTOS

TESTE ESTRUTURAL DE APLICAÇÃO DE BANCO DE DADOS EM PROGRAMA JAVA

MARÍLIA
2008

NÍVIA TOGNI CÊGA DOS SANTOS

TESTE ESTRUTURAL DE APLICAÇÃO DE BANCO DE DADOS EM PROGRAMA JAVA

Trabalho de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharelado em Ciência da Computação.

Orientador:
Prof. Dr. EDMUNDO SERGIO SPOTO

MARÍLIA
2008

SANTOS, Nívia Togni Cêga dos
Aplicação de Tecnicas de Teste Estrutura em um sistema de banco de dados Java/ Nívia Togni Cêga dos Santos; orientador: Edmundo Sergio Spoto. Marília, SP: [s.n.], 2008.
71f.

Trabalho de Curso (Graduação em Bacharelado em Ciência da Computação) - Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM.

1. Teste de Software 2. Teste Estrutural Orientado a Objetos

CDD: **005.14**

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Nívia Togni Cêga dos Santos

TESTE ESTRUTURAL DE APLICAÇÃO DE BANCO DE DADOS EM PROGRAMA JAVA

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 7,5 (Sete e meio)

Orientador: Edmundo Sérgio Spoto

1°. Examinador: Valter Vieira de Camargo

2°. Examinador: Paulo Augusto Nardi

Marília, 13 de novembro de 2008.

AGRADECIMENTOS

Agradeço principalmente ao meu orientador Profº Dino pelo direcionamento, apoio, pela paciência e pelo tempo dedicados a mim e a este trabalho.

Agradeço também ao professor Nardi sempre ajudando quando preciso, pela paciência demonstrada, pelos ensinamentos passados e pelas orientações tão necessárias para a conclusão deste trabalho.

Ao meu namorado Bruno, pela compreensão e apoio nas horas difíceis e por me mostrar o caminho da persistência até conseguir atingir os objetivos.

Aos meus pais, pelo carinho e compreensão.

Ao meu amigo Diego Roberto pelo apoio e serviços prestados quando eu mais precisava, fundamentais para a realização deste trabalho.

Aos outros professores pelos ensinamentos trasmitidos durante o ano.

Aos meus outros amigos pela força, pelas brincadeiras e pelo companheirismo que existiu durante esta jornada.

SANTOS, Nívia Togni Cêga dos. **Teste Estrutural de Aplicação de Banco de Dados em Programa Java.** 2008**.** 71f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2008.

RESUMO

Este trabalho verifica se um dos critérios de teste que foram definidos para banco de dados relacionais podem ser utilizados em Sistemas Orientados a Objetos com uso de banco de dados relacional. Um estudo de caso será criado visando exercitar os principais critérios estruturais de teste de ABDR em programas Orientado a Objetos. A partir dos resultados obtidos na etapa de teste estrutural foi realizada uma análise dos tipos de defeitos que são observados em cada critério levando em consideração as características do Banco de Dados Relacional. Finalizando foi apresentada uma conclusão final dos resultados obtidos.

Palavras-Chave: Teste de Software, Teste Estrutural Orientado a Objetos.

SANTOS, Nívia Togni Cêga dos. **Teste Estrutural de Aplicação de Banco de Dados em Programa Java.** 2008**.** 71f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2008

ABSTRACT

The objective of this paper is to verify whether the one of the test criteria that were set for relational database systems can be used in the object oriented with a relational database. A case study will be created targeting exercise the main criteria for structural testing ABDR in the object oriented programs. From the results obtained in the test phase of structural intends to carry out an analysis of the types of defects that are observed in each test taking into account the characteristics of the relational database.

Keywords: Software Engineer, Structural Test Objected Oriented.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

INTRODUÇÃO

Segundo Pressman (2000) teste de software é uma das técnicas de obtenção da qualidade de software. O teste de software é composto por três técnicas: Funcional (caixa preta), quando os requisitos são obtidos a partir da especificação; Estrutural (caixa branca) quando derivam-se os requisitos a partir dos aspectos de implementação do software; e baseado em erros, a obtenção dos elementos requeridos para se fazer o teste provém de semear erros sutis no programa original otendo programas mutantes a ser testados(Delamaro, 2007).

Segundo Myers (1988), o teste de software é a ação de executar o software com um dado de entrada e obter uma saída. A saída obtida é comparada com a saída esperada (com base na especificação) e essas informações são denominadas de *Casos de Teste*.

Toda etapa de teste deve ser devidamente planejada, levando-se em consideração a geração dos casos de testes e os respectivos elementos requeridos que se pretende exercitar. Porém o projeto de casos de testes pode ser tão difícil quanto fazer um programa, mas é necessário que se faça, pois para se ter um programa considerado com qualidade o teste de software é fundamental (Pressman, 2000). O teste de software mostra os erros existentes no programa para que se possa corrigir antes da comercialização e assim garantir a confiabilidade do produto.

Devido a grande importância do teste de software, vários critérios e ferramentas de apoio foram criados e estão sendo desenvolvidas para ajudar o teste de software.

O teste de software é uma das principais direções para a área de teste de software. Porém com o surgimento da linguagem orientada a objetos, alguns desafios foram lançados ao teste de software, pois algumas das características essenciais para a orientação a objetos causam um grande impacto no teste de software. Entre elas estão o encapsulamento, polimorfismo e herança (Harrold, 2000 apud Domingues, 2002).

Neste trabalho serão apresentados casos de teste projetados para um banco de dados relacional e que será usado para testar uma aplicação OO. Estratégias estruturais de teste serão aplicadas visando exercitar a eficácia do critério em relação à detecção de defeitos. Dessa forma pode-se verificar se os testes existentes foram suficientes para cobrir todos os elementos requeridos na tentativa de detecção de novos defeitos no programa.

OBJETIVOS

O objetivo deste trabalho é verificar se os critérios de teste de integração baseado no fluxo de dados das tabelas definidos para ABDR em aplicações procedimentais podem ser adaptados para Sistemas OO com uso de banco de dados relacional. Um estudo de casos será criado visando exercitar o critério *todos-t-usos-intra* de (SPOTO, 2000) em uma ABDR em programa OO.

A partir dos resultados obtidos na etapa de teste estrutural pretende-se realizar uma análise dos tipos de defeitos que são observados em cada critério levando em consideração as características do Banco de Dados Relacional.

MOTIVAÇÃO

A Engenharia de Software visa construir produtos com baixo custo e boa qualidade. Para isso várias etapas de desenvolvimento devem possuir apóio que auxiliam na obtenção da qualidade. A Etapa de teste de software sofre muitas vezes penalidades por ser uma etapa mal projetada e por falta de ferramentas específicas na geração de técnicas de teste. Com isso a geração de teste estrutural baseada nas características do banco de dados relacional pode trazer benefícios na correção de tipos de defeitos.

Este trabalho pretende mostrar que algumas técnicas de teste estrutural baseada na ABDR podem ser úteis na detecção de defeitos mesmo quando a aplicação for Orientada a Objetos.

ORGANIZAÇÃO DO TRABALHO

No Capitulo 1 são apresentados os conceitos de aplicações orientadas a objetos descrevendo os conceitos básicos de: objetos, classes, herança, polimorfismo, encapsulamento, etc; São apresentados ainda os conceitos sobre testes de software para aplicações OO, para que se possam entender como essas características básicas da linguagem OO influenciam no teste de software.

No Capitulo 2 são apresentados conceitos sobre técnicas de teste estruturais, quais as mais utilizadas, como as técnicas de teste funcionais, estruturais e baseadas em erros.

No Capítulo 3 descreve o critério de teste todos-t-uso intra-classe utilizado neste trabalho. No Capitulo 4 são apresentados os estudos de caso, a análise sobre o programa que é testado. É apresentado o diagrama de classes e separadamente, a apresentação das classes do software em teste e o critério intra-classe que utilizado neste trabalho. Também são descritos:

os elementos requeridos do critério *todos-t-uso-intra*, os repectivos casos de testes para exercitar os elementos requeridos definidos e os principais resultados obtidos.

E finalizando com uma conclusão do trabalho e a apresentação de possíveis trabalhos futuros.

CAPITULO 1 - APLICAÇÃO ORIENTADA A OBJETOS

Neste capitulo são apresentados os conceitos de orientação a objetos e características chaves deste tipo de linguagem para que seja entendido o grande impacto que causa a orientação a objeto em testes de software.

1.1. Definição e Conceitos básicos

Segundo Coad (1993), a orientação a objetos (OO), é um paradigma de análise, projeto e programação de sistemas de *software* baseado na composição e interação entre diversas unidades de software chamadas de objetos.

Em 1960, derivada da liguagem Algol, surgiu a linguagem Simula criada por Ole Johan Dahl e Kristen Nygaard, quando apareceu pela primeira vez os conceitos de classes, rotinas correlatas e subclasses.

Na Tabela 01 são mostrados os principais acontecimentos da história de orientação a objetos.

Tabela 01: Histórico de programas orientado a objetos

1967	**Simula-67**
1972	Artigo de Dahl sobre ocultamento da informação
1976	primeira versão de Smaltalk
1983	Primeira versão de C++
1988	primeira versão de Eiffel
1995	Primeira versão de Java

Ainda segundo Coad (1993), a aplicação Orientada a Objetos procura fazer com que sejam semelhantes os objetos da computação com os objetos do mundo real. A idéia de programas simulando o mundo real surgiu com Simula-67, cujo objetivo era a construção de aplicações de simulação .

Já o termo Programação Orientada a Objetos foi criado por Alan Kay, autor da linguagem de programação Smalltalk (COAD, 1993).

Segundo Deitel(2005), o projeto orientado a objetos (OOD) modela objetos do mundo real. O autor diz que ele se aproveita das relações de classe, nas quais os objetos de uma certa classe tem as mesmas carecterísticas. A orientação a objetos tira proveito de relações de herança e até herança múltipla, em que as classes de objetos recém-criadas são

derivadas aproveitando as características das classes existentes e adcionando-se as características próprias.

> O projeto orientado a objetos (OOD) oferece uma maneira mais natural e intuitiva para visualizar o processo de projeto – a saber, modelando objetos do mundo real, seus atributos e seu comportamento. O OOD também modela a comunicação entre objetos. Da mesma forma que pessoas enviam mensagens umas as outras (por exemplo, o sargento que ordena ao soldador para se manter em posição de sentido) os objetos também se comunicam através de mensagens (DEITEL, 2005).

Temos várias definições de objeto. Segundo Coad(1993), "um objeto é qualquer coisa, real ou abstrata, a respeito da qual armazenamos dados e os métodos que o manipulam." Já para Odell(1995), um objeto é "[...] qualquer coisa real ou abstrata, a respeito da qual armazenamos dados e os métodos que os manipulam [..]." Ele diz ainda que este objeto pode ser composto de outros objetos. Esse objeto possui características e serviços. De acordo com Deitel(2005), objetos diferentes podem ter atributos semelhantes e podem exibir comportamentos semelhantes.

Coad(1993) também diz que um objeto pode ser composto de outros objetos e estes objetos podem ser compostos de mais outros objetos. Dessa forma, esta estrutura permite que objetos complexos possam ser definidos por outros objetos mais simples..

Ricarte (2001) define objeto como sendo uma entidade do mundo real que tem uma identidade. Objetos podem representar entidades concretas (um arquivo no computador, uma bicicleta) ou entidades conceituais (uma estratégia de jogo, uma política de escalonamento em um sistema operacional). De acordo com Ricarte (2001), quando se diz que cada objeto tem sua identidade significa que dois objetos são diferentes, únicos, mesmo que eles apresentem exatamente as mesmas caraterísticas.

No dicionário especificamente para OO:

> Objeto: é uma abstração de alguma coisa no domínio de um problema e sua implementação, refletindo a capacidade de um sistema para manter informações sobre ela, interagir com ela, ou ambos: um encapsulamento de valores de atributo e seus serviços exclusivos (Webster's, 1977).

De acordo com Sommerville (2004) , um objeto possui um comportamento, um estado e uma identidade. O estado é como se encontra as informações encapsuladas (escondidas) pelo objeto. O comportamento define o modo que um objeto age e reage em termos das suas mudanças de estado e é completamente definido pelas suas operações e a indentidade é a propriedade de um objeto que o distingue de outro objeto.

Durante uma aplicação OO, os objetos podem ser criados, excluídos, modificados e executar ações. Podem ser temporários ou durar para sempre até que seu desaparecimento seja ordenado, o que chamamos de objetos persistentes. Por isso diz-se que objeto é algo dinâmico (PRESSMAN, 1995).

O objeto então pode-se dizer que é um conjunto de informações (atributos) que executam operações (métodos) (DEITEL, 2005). Os métodos indicam como os dados do objeto vão ser manipulados e referenciam apenas a estrutura de dados deste objeto

Esses objetos se comunicam através de "mensagens", que são os métodos de cada objeto. Todos os objetos que se assemelham na troca de mensagens são agrupados em classe (COAD,1993).

De acordo com Ricarte(2001), a abordagem de orientação a objetos favorece a aplicação de diversos conceitos considerados fundamentais para o desenvolvimento de bons programas, tais como abstração e encapsulação. Ele ainda afirma que tais conceitos não são exclusivos desta abordagem, mas são suportados de forma melhor no desenvolvimento orientado a objetos do que em outras métodologias.

A abstração é essencial para a aplicação. É atraves dela que os métodos são definidos, os atributos são atribuídos e características são dadas.

Oxford (1986) apresenta uma definição de abstração que diz:

> "Abstração: O principio de ignorar os aspectos de um assunto que não sejam relevantes para o proposito em questão, a fim de dedicar uma concentração maior aos aspectos relevantes."

Ou seja, abstração é focar somente naquilo em que temos interesse, para assim podermos definir corretamente atributos e métodos. Através da abstração pode-se mapear um objeto do mundo real e abstrair as ações, características, atributos essenciais referentes a esses objetos e transformá-los em um software.

De acordo com Odell (1995), a abstração é uma maneira de administrarmos a complexidade dos objetos do nosso mundo,

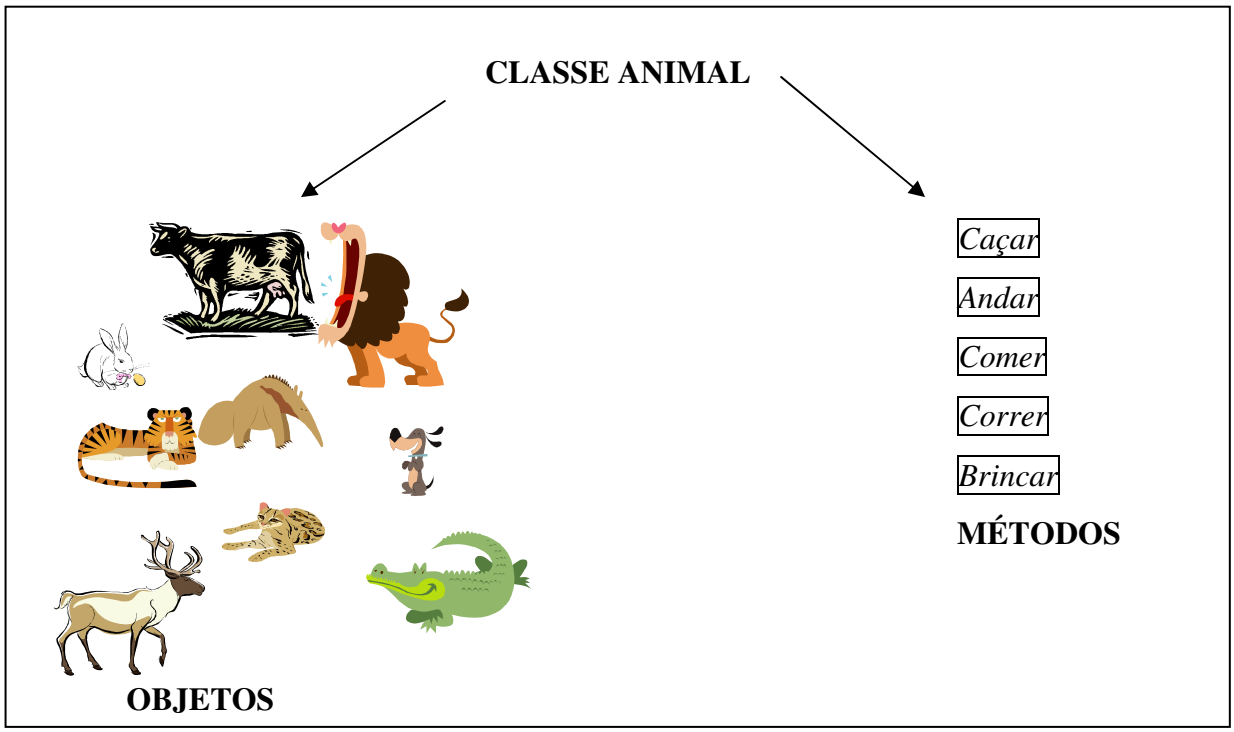
Um atributo de um objeto são as característcas do objeto. Os atributos só podem ser manipulados pelas operações daquele objeto. Cada objeto possui seu próprio conjunto de atributos. As operações são os métodos que modificam os atributos do objeto e podem atuar sobre um objeto ou uma classe de objetos. Cada objeto possui seu próprio conjunto de métodos. Os métodos são invocados por "mensagens" (SILVA, 2008).

Ainda de acordo com Silva (2008), o objeto tem sua parte compartilhada, a interface. As “mensagens” movem-se através da interface e dizem qual operação do objeto é a desejada, mas não especificam como devem ser executadas. O objeto ao receber a mensagem, vai determinar como as operações devem ser executadas .

Agora, quando temos vários objetos semelhantes, podemos agrupá-los. Quando temos vários objetos agrupados, chamamos de classe.

Classe é um conjunto de objetos similares do mundo real. Todo objeto é uma instância de uma classe (VICENZI, 2004). A instância são os valores de cada objeto e é pelos valores de cada objeto que é possível distinguir cada um deles.

Figura 1: Exemplo de classes abstratas



Deitel (2005) diz que as classes consistem em partes chamadas métodos que realizam tarefas e retornam informações quando elas completam suas tarefas.

As classes podem ser abstratas, quando servem como um modelo e não são instanciadas. Podem ser persistentes, e, como o próprio nome diz, persistem do início ao fim da aplicação ou podem ser transientes, quando são utilizadas somente quando são chamadas e utilizadas (BUZATO, 1998). Na Figura 1 foi mostrado o exemplo de uma classe abstrata.

As classes podem relacionar-se entre si. Um relacionamento de associação é um relacionamento que objeto tem que ter com outro para poder cumprir com suas tarefas.

O agrupamento de objetos em classes é um poderoso mecanismo de abstração. Desta forma, é possível generalizar definições comuns para uma classe de objetos, ao invés de repetí-las para cada objeto em particular (RICARTE, 2001).

De acordo com Pressman (2000), uma aplicação OO "cria uma representação do domínio do problema do mundo real e leva-a a um domínio de solução que é o software." Ele afirma ainda que, ao contrário de outros paradigmas, o projeto orientado a objetos resulta num projeto que interliga objetos de dados (itens de dados) e operações de processamento de uma forma que modulariza a informação e o processamento, não apenas o processamento.

Há alguns outros elementos chaves em uma aplicação OO. São eles: encapsulamento de dados, polimorfismo, e herança.

De acordo com Odell(1995), encapsulamento é o ato de empacotar ao mesmo tempo dados e métodos. Isso é feito para permitir apenas que os métodos do objeto em questão possa acessar os dados. De uma forma menos complexa, Vicenzi(2004) define encapsulamento como sendo a capacidade do objeto de impedir que outros objetos tenham acesso aos seus métodos. O encapsulamento de dados visa esconder detalhes de implementação. Assim, o programador pode alterar dados de um método sem que este altere seu comportamento.

Oxford (1986) definiu Encapsulamento como sendo:

> "Encapsulamento é ocultação da informação. Um princípio usado no desenvolvimento de uma estrutura global de programa, segundo o qual cada componente de um programa deve encapsular ou ocultar uma só decisão de projeto. A interface para cada modulo é definida por de modo a revelar o mínimo possível sobre seu funcionamento interno."

Deitel (2005) confirma a afirmação acima dizendo:

> "O Desenvolvimento Orientado a Objeto (DOO) encapsula dados (atributos) e funções (comportamento) em objetos; os dados e as funções de um objeto estao intimamente amarrados. Os objetos tem a propriedade de de ocultamento de informações. "

Isto significa que mesmo que os objetos possam saber como se comunicar uns com os outros através de interfaces bem-definidas, normalmente não é permitido aos objetos saber como outros objetos são implementados, ou seja, os detalhes da implementação ficam escondidos dentro dos proprios objetos (DEITEL, 2005).

Os objetos encapsulam seus atributos, que só são acessíveis pelos métodos da classe. Para que outras classes acessem estes dados, devem invocar os métodos públicos. A vantagem

disso é o reuso, a legibilidade e a manutenibilidade. O ocultamento de informações facilita a modificação do programa e simplifica a percepção de classe pelo cliente (COAD,1993).

Odell (1995) diz "[...] O encapsulamento é importante porque separa a maneira como um objeto se comporta da maneira como ele é implementado".Significa que isso permite, de acordo com Odell (1995), que as implementações dos objetos sejam modificadas sem exigir que os aplicativos que as usam sejam também modificadas.

E o conceito de herança, de acordo com DEITEL (2005):

> É uma forma de reutilização de software em que novas classes são criadas a partir de classes existentes, absorvendo seus atributos e comportamentos e sofisticando-os com capacidades que as novas classes exigem.

Ou seja, a herança é a capacidade de se poder construir uma nova classe (chamada subclasse) a partir de uma classe já existente (chamada superclasse). Com isso a subclasse herda as características comuns da superclasse, como métodos e atributos e passam a fazer parte da subclasse também. Não que a subclasse só possa possuir os métodos e atributos herdados. Ela pode adicionar novos métodos e atributos também. Porém, a subclasse não pode acessar diretamente os métodos private de sua superclasse (DEITEL, 2005).

> A herança tira proveito dos relacionamentos entre classes, nos quais os objetos de uma certa classe – como uma classe de veículos – tem as mesmas caracteristicas. As classes de objetos recem criadas derivam-se ao absorver as caracteristicas das classes existentes e ao adcionar suas proprias caracteristicas (DEITEL, 2005).

Quando se cria uma nova classe, ao invés de reescrever totalmente novas variáveis de instância e métodos de instâncias, pode-se determinar que a nova classe deve herdar as variáveis e os métodos de uma superclasse definida anteriormente. A nova classe é chamada de subclasse e pode se tornar uma superclasse para alguma subclasse futura (DEITEL,2005).

Ainda de acordo com Deitel (2005), "uma superclasse direta de uma subclasse é aquela da qual a subclasse herda explicitamente (através da palavra-chave *extends*). Uma superclasse indireta é herdada de uma ou mais níveis acima da hierarquia da classe."

Cada objeto de uma subclasse também é um objeto da superclasse daquela classe. Porém, mesmo que um objeto de uma subclasse possa ser considerado como um dos tipos da sua superclasse, os objetos da superclasse não são considerados todos os tipos da subclasse, afirma Deitel (2005).

Um dos problemas da herança é que a subclasse pode herdar métodos que não precisa ou que não deveria ter. Fica sob responsabilidade do programador assegurar que os

métodos fornecidos pela superclasse sejam apropriados também para a subclasse, de acordo com DEITEL (2005).

Ricarte (2001) diz que uma operação polimôrfica ocorre quando uma mesma operação pode se comportar de forma diferente em classes diferentes. De acordo com ele, um método é uma implementação específica de uma operação para uma certa classe.

Para Vicenzi (2004), polimorfismo é a redefinição de um método em classes diferentes. Este método possuirá também comportamento diferente. Assim, o objeto pode assumir diferentes formas, ou seja, o mesmo nome de um método, porém com funções diferentes. Dessa forma, ao receber uma mensagem, um objeto pode ativar varios métodos ao mesmo tempo. De acordo com Deitel(2005), o polimorfismo permite-nos escrever programas de forma geral, para tratar de uma grande variedade de classes existentes ou ainda a serem especificadas, além de tornar fácil adcionar novos recursos a um sistema. Ele ainda afirma que com o polimorfismo, "é possivel projetar e implementar sistemas que são mais facilmente extensíveis."

Ricarte (2001) ainda afirma que polimorfismo também implica que uma operação de uma mesma classe pode ser implementada por mais de um método. O usuário não precisa saber quantas implementações existem para uma operação, ou explicitar qual método deve ser utilizado: a linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumentos para a operação. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos.

Agora que os conceitos foram apresentados, vamos conhecer quais são os maiores problemas nos testes de aplicação orientada a objetos.

1.2. Problemas nos testes de Aplicação OO.

De acordo com Vicenzi (2004), "o paradigma de programação OO, possui um conjunto de construções poderosas, que apresentam risco de defeito e problemas de teste[...]." Isto acontece devido ao encapsulamento de métodos e de atributos dentro de uma classe, pela variedade de modos como um subsistema pode ser composto e da possibilidade de, em poucas linhas de código, dar um comportamento ao sistema que só será definido em tempo de execução (acoplamento dinâmico).

Usando como exemplo uma classe com herança múltipla, com algumas superclasses e todas contribuem na hieraquia e com muitos métodos polimórficos. De acordo com

Vincenzi(2004) o desenvolvedor deve garantir que todos os métodos das superclasses funcionem adequadamente na subclasse e que não ocorra nenhuma interação indesejável entre os métodos. Devido ao polimorfismo e o acoplamento dinâmico, aumenta drasticamente o número de caminhos que devem ser testados e o encapsulamento pode criar obstáculos que limitam a visibilidade do estado dos objetos (VICENZI, 2004).

Deve-se entender também a questão da reusabilidade. Os componentes do software que são disponibilizados para reúso devem ser altamente confiáveis e por isso o teste extensivo é exigido para se obter um reúso efetivo (VICENZI, 2004).

Vicenzi (2004) finaliza dizendo que embora a programação orientada a objetos reduza a ocorrência de alguns defeitos que ocorriam na programação procedimental, por outro lado ela aumenta as chances de ocorrência de outros tipos de defeitos. Como um método possui poucas linhas de código, é menos provável que aconteça um defeito de fluxo de controle. Mas erros de grafia incorreta ou sintaxe são tão comuns quanto antes. Tão comuns quanto defeitos na programação da interface de programas procedimentais. Ele afirma que, como programas OO tem muitos métodos e muitas interfaces aumenta a ocorrência deste tipo de defeito. Além destes tipos de defeito, podem ocorrer defeitos devido ao encapsulamento, a herança, as classes abstratas e genéricas, polimorfismo, acoplamento dinâmico e outros.

1.3. Liguagem Java

A linguagem Java é uma linguagem de programação orientada a objetos, desenvolvida por uma equipe de pessoas na Sun Microsystems. Inicialmente elaborada para ser a linguagem-base de projetos de software para produtos eletrônicos, mas em 1995 ficou mundialmente conhecida graças a World Wide Web (SILVEIRA, 2003). Possui linguagem de alto nível, com sintaxe extremamente similar à do C++, e com diversas características herdadas de outras linguagens, como Smalltalk e Modula-3.

Ao contrário de C++, que é uma linguagem híbrida, Java é uma linguagem orientada a objetos que segue a linha purista iniciada por Smalltalk. Com a exceção dos tipos básicos da linguagem (*int*, *float*, etc.), a maior parte dos elementos de um programa Java são objetos. O código é organizado em classes, que podem estabelecer relacionamentos de herança simples entre si. Somente a herança simples é permitida em Java (SILVEIRA, 2003).

Até hoje, a plataforma Java já atraiu mais de 5 milhões de desenvolvedores de software. Ela é usada em todos os principais setores e está presente em uma ampla gama de dispositivos, computadores e redes (FONTE: http://java.com/pt_BR/about/).

CAPITULO 2 – TESTE DE SOFTWARE

Nas últimas duas décadas os profissionais da área de Engenharia de Software têm demonstrado uma grande preocupação em incorporar qualidade no processo de desenvolvimento de seus produtos de software (SPOTO, 2000).

Segundo Pressman (2000), o processo de desenvolvimento de software se divide em três fases genéricas: definição, desenvolvimento e manutenção. A fase de definição é a identificação dos requisitos básicos do sistema e do software. A fase de desenvolvimento define como os requisitos do sistema serão satisfeitos e dentro desta fase ocorrem três etapas distintas: projeto, codificação e teste de software. A fase de manutenção concentra-se nas modificações feitas sobre o software devido às atividades como: correção, melhoria e adaptação. Este trabalho concentra-se na etapa de teste de software.

Spoto (2000) afirma que existem muitas áreas de desenvolvimento de software em que as iniciativas e o esforço têm sido muito incipientes. Em particular, ele cita que existe muito pouco quando se trata de suporte de teste de programas de ABDR, mesmo no âmbito internacional.

> Existem vários tipos de aplicação que carecem de técnicas apropriadas para o teste de software. Uma delas é o desenvolvimento de programas de Aplicações de Banco de Dados Relacional (ABDR) que utiliza a linguagem de consulta SQL embutida em linguagens procedimentais tais como: C, Pascal, PL/I, etc (SPOTO, 2000).

O teste é uma importante ferramenta de feedback e fornece uma base para a interação com os participantes no projeto. Devido a crescente complexidade dos softwares, não é surpresa o fato de que cerca de 30% a 50 % do orçamento destinado ao desenvolvimento do software seja gasto diretamente com testes (PETERS, 2001).

De acordo com Myers(1976), “teste é o processo de execução de um programa com a intenção de encontrar erros”.

De acordo com Spoto(2000), o projeto de teste de software tem as seguintes atividades: planejamento, projeto de casos de teste e execução e avaliação dos resultados dos testes. Os métodos utilizados, ainda de acordo com ele são: baseados essencialmente nas técnicas funcional, estrutural e baseada em defeitos.

Convêm observar que este trabalho concentra-se em teste de software baseado na técnica estrutural.

2.1 Técnicas de Teste

Nesta seção serão apresentadas algumas das técnicas de teste existentes. Existem três técnicas de teste: funcional (caixa preta), estrutural (caixa branca) e baseada em erros. As etapas de teste se dividem em: teste de unidade, teste de integração e teste de sistema.

O teste funcional tem como premissa tratar o software como uma caixa que não se conhece o conteúdo, só é possível visualizar o lado externo .Desse modo, é necessário utilizar essencialmente a especificação funcional do programa para derivar os casos de teste que serão empregados sem se importar com os detalhes de implementação (BEIZER, 1990 apud VICENZI, 2004). Assim, uma especificação correta e de acordo com os requisitos do usuário é essencial para esse tipo de teste (VICENZI, 2004). Ou seja, a técnica funcional visa a estabelecer requisitos de teste derivados da especificação funcional do software (teste caixa preta) enquanto a técnica estrutural apóia-se em informações derivadas diretamente da implementação (teste caixa branca) (SPOTO, 2000).

O teste estrutural apóia-se nas informações derivadas diretamente da implementação do código (NARDI, 2006). Este tipo de teste visa caracterizar um conjunto de componentes elementares do software que deve ser exercitado, de acordo com Spoto (2000).

Este tipo de teste é conhecido como caixa branca e baseia-se num minucioso exame de detalhes procedimentais (PRESSMAN, 1995). São testados os caminhos lógicos, através do fornecimento de casos de teste que irá por a prova conjuntos específicos de condições. Para Pressman (1995), poderia parecer que este tipo de teste levaria a 100% de programas corretos e só nos restaria desenvolver os casos de teste, executá-los e analisar os resultados. Porém, ele afirma que o teste exaustivo, causaria grandes problemas logísticos, pois o número de caminhos possíveis pode ser enorme, mesmo para um programa pequeno.

Como este trabalho é baseado neste tipo de teste, ele será discutido com maior profundidade nos capítulos que seguem.

Já o teste baseado em defeitos utiliza informações sobre os defeitos típicos de desenvolvimento para derivar os requisitos de teste (MORREL, 1990 apud NARDI, 2006).

Delamaro (1995) definiu um conjunto de operadores de mutação para a linguagem C, gerando assim vários mutantes a partir desses operadores e uma ferramenta para exercitar esse tipo de teste (Ferramenta Proteum).

2.1.1 Teste de unidade

O teste de unidade é a etapa de teste individual, em que são testados cada procedimento ou função isoladamente, de um programa.

O teste de unidade para programas OO também é chamado de teste intra-método (HARROLD & ROTHERMEL, 1994). Por definição, uma classe engloba um conjunto de atributos e métodos que manipulam esses atributos. Assim sendo, considerando uma única classe, já é possível pensar-se em teste de integração.

O teste de unidades concentra-se no esforço de verificação da menor unidade de projeto de software – o módulo. Usando a descrição do projeto detalhado como guia, caminhos de controle importantes são testados para descobrirem erros dentro das fronteiras do módulo. A complexidade relativa dos testes e os erros detectados como resultado deles é limitado pelo campo de ação restrito estabelecido para o teste de unidade. O teste de unidade baseia-se sempre na caixa branca, e esse passo pode ser realizado em paralelo (PRESSMAN, 2000).

2.1.2 Teste de integração

O teste de integração é o teste que se aplica após o teste de cada unidade. Nesta etapa do teste procura-se integrar todos os módulos de um programa até que todas unidades (procedimentos e funções) estejam integradas. No caso de programas OO pode-se considerar a menor unidade como sendo um método ou como sendo uma classe.

Métodos da mesma classe podem interagir entre si para desempenhar funções específicas, caracterizando uma integração entre métodos que deve ser testada: teste inter-método (HARROLD & ROTHERMEL, 1994).

Alguns autores entendem que a classe é a menor unidade no contexto de software OO (ARNOLD & FUSON, 1994; BINDER, 1999; MCDANIEL & MCGREGOR, 1994; PERRY & KAISER, 1990). Nessa direção o teste de unidade poderia envolver o intra-classe e, o teste de integração corresponderia ao teste inter-classe (HARROLD & ROTHERMEL, 1994).

Vicenzi (2004) diz que pode se considerar que em programas OO, a menor unidade a ser testada é um método sendo que a classe a qual o método pertence pode ser vista como o driver do método. Sem a existência da classe não é possível executar um método. No paradigma procedimental, o teste de unidade também é chamado de intraprocedimental e no paradigma orientado a objetos intra-método (HARROLD & ROTHERMEL, 1994).

O teste de Integração é uma técnica sistemática para a construção da estrutura do programa, realizando-se, ao mesmo tempo, testes para descobrir erros associados a interfaces. O objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa que foi determinada pelo projeto (PRESSMAN, 2000).

De acordo com Pressman (2000), o objetivo da fase de integração é encontrar falhas provenientes da integração interna dos componentes de um sistema. Essas interfaces são testadas na fase de teste de sistema, apesar de, a critério do gerente de projeto, estas interfaces podem ser testadas mesmo antes de o sistema estar plenamente construído.

Harrold & Rothermel (1994) definem ainda outros dois tipos de teste de integração para programas OO: teste intra-classe e teste inter-classe.

2.1.2.1 Intra-classe

No teste intra-classe é testada a interação entre os métodos públicos da mesma classe. Para isso são feitas chamadas ao método mas em seqüências diferente para verificar como o método irá se comportar diante disso (SPOTO,2000). Ou seja, no teste intra-classe são testadas interações entre métodos públicos fazendo chamada a esses métodos em diferentes seqüências. O objetivo é identificar possíveis seqüências de ativação de métodos inválidas que levem o objeto a um estado inconsistente (HARROLD & ROTHERMEL, 1994).

2.1.2.2 Inter-classe

No teste inter-classe acontece o mesmo que no teste intra-classe falado acima, porém os métodos não precisam estar na mesma classe (SPOTO, 2000). Ou seja, no teste inter-classe o mesmo conceito de invocação de métodos públicos em diferentes seqüências é utilizado, entretanto, esses métodos públicos não necessitam estar na mesma classe (HARROLD & ROTHERMEL, 1994).

2.1.3 Teste de sistema

O teste de sistema é um conjunto de testes diferentes, com a finalidade principal de testar o sistema completo de software. Isso significa que, depois de realizados os testes de unidade, integração e validação os quais têm suas formas de testar diferentes, deve-se verificar se os elementos do sistema foram adequadamente integrados e realizam as funções atribuídas (PRESSMAN, 2000).

2.2 Teste de Software de Aplicação OO utilizando BDR

Por mais que a abordagem orientada a objetos nos apresente várias vantagens, a realização dos testes é um grande problema. Ao mesmo tempo que algumas características desse tipo de linguagem diminui a ocorrência de erros, podem ocorrer novos erros no código. Não é só porque software foi implementado em linguagem OO que não deva ser testado. É claro que apesar da orientação a objeto apresentar uma melhor arquitetura de sistemas, ela não impede que ocorram erros de sistemas.

Algumas das vantagens que a linguagem OO proporciona, de acordo com Martins (2003), são : redução no tamanho dos métodos através da utilização de heranças; algoritmos menos complexos; facilidade na localização e correção de defeitos; encapsulamento.

A orientação a objeto traz novos desafios ao teste de software, pois apresenta tipos diferentes de erros e por isso, uma abordagem especial é necessária. Os conceitos de herança, polimorfismo, encapsulamento de dados e acoplamento dinâmico causam um impacto no teste de software convencional. Às vezes técnicas que são bem aceitas em um determinado tipo de software é inadequado para um software OO (MCDANIEL & MCGREGOR, 1994).

Num software OO, a parte mais difícil é o teste. As técnicas tradicionais não são suficientes para detectar erros. Essas técnicas devem ser extendidas para conseguir abranger os conceitos descritos acima (MCDANIEL & MCGREGOR, 1994).

Devido a grande diversidade de critérios que tem sido estabelecidos e reconhecido o caráter complementar das técnicas e critérios de teste, um ponto crucial que se coloca nessa perspectiva é a escolha e/ou a determinação de uma estratégia de teste, que em última análise passa pela escolha de critérios de teste, de forma que as vantagens de cada um desses critérios sejam combinadas objetivando uma atividade de teste de maior qualidade. Estudos teóricos e empíricos de critérios de teste são de extrema relevância para a formação desse conhecimento, fornecendo subsídios para o estabelecimento de estratégias de baixo custo e alta eficácia (MALDONADO , 1998).

O teste estrutural em aplicações OO é diferente em relação aos outros. O objetivo neste caso é verificar se a estrutura interna da unidade, e seus caminhos internos estão corretos. Por causa do polimorfismo, o teste é mais complexo, pois quando uma “mensagem” é enviada, qualquer classe pode recebê-la e assim não é possivel ver no código qual classe que receberá a mensagem (BUZATO, 1998).

> "(...) sugeri que havia duas atividades fundamentais no processos de teste. Esses são os testes de componentes , em que os componentes do sistena são testados individualmente, e os testes de integração , em que coleções de componentes são integradas em subsistemas e no sistema final para teste. Essas atividades são igualmente aplicáveis a sistemas orientados a objetos. Contudo, existem importantes diferenças entre os sistemas orientado a objetos e os sistemas desenvolvidos utilizando-se um modelo funcional" (SOMMERVILE,2004).

As diferenças que Sommervile (2004) diz acima seguem abaixo:

1. Objetos como componentes individuais são, muitas vezes, maiores do que funções isoladas.

2. Os objetos que são integrados em subsistemas são, em geral, indevidamente acoplados, e não há um nível superior óbvio para o sistema.

3. Se os objetos forem reutilizados, os testadores podem não ter nenhum acesso ao código-fonte do componente para análise.

Isso significa, ainda de acordo com Sommervile(2004), que as abordagens dos testes estruturais devem ser ampliadas para cobrir objetos de maior granularidade e que abordagens alternativas de teste de integração podem ser adotadas.

De acordo com Sommervile (2004) quatro níveis de teste podem ser identificados:

1. Testar as operações individuais associadas com objetos. São funções ou procedimentos, podendo ser utilizadas as técnicas de caixa preta e caixa branca.

2. Testar classes de objetos individuais. Os princípios de teste de caixa preta devem ser estendidos para cobrir seqüências de operação relacionadas.

3. Testar agrupamento de objetos. Deve ser utilizadas técnicas com base em cenário, pois o teste de integração top-down ou bottom-up seria inadequada.

4. Testar o sistema orientado a objetos. Verificação e validação, feita normalmente como qualquer outro tipo de sistema.

Neste trabalho, será utilizada a técnica de teste estrutural para a realização dos testes de um software OO. De acordo com Nardi (2006) "[...] ao se aplicar a técnica de teste estrutural, faz-se necessário conhecer a implementação do código.[...]". É necessário este conhecimento pois a técnica estrutural utiliza a estrutura interna do programa para derivar os casos de teste. Após conhecer a estrutura interna do programa é necessário escolher os critérios de testes a serem utilizados e que serão apresentados a seguir.

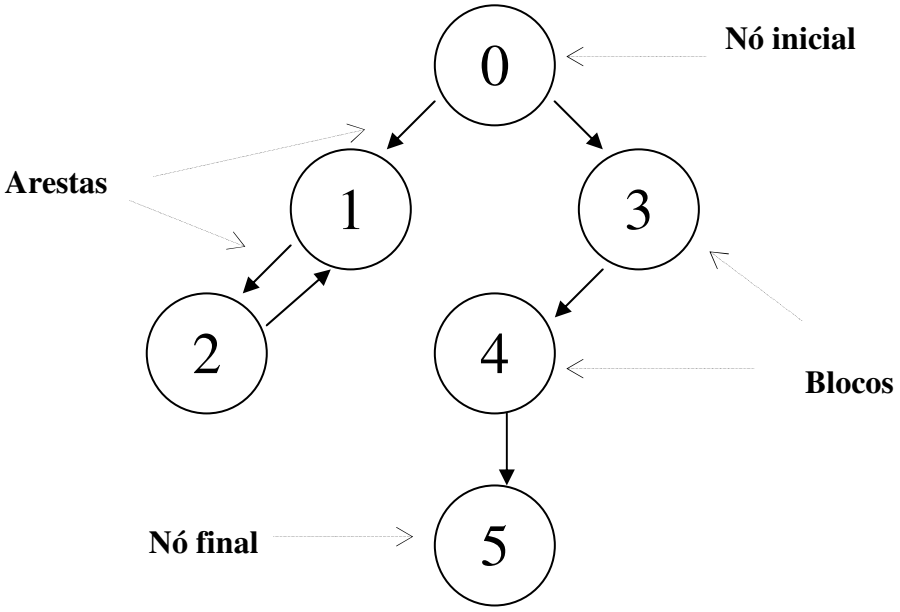
2.3 Critérios de Teste Estrutural

De acordo com Vincenzi(2004), o teste estrutural (conhecido também como caixa branca) leva em consideração os aspectos de implementação na escolha dos casos de teste. A maioria destes critérios utiliza uma representação de programa conhecida como grafo de fluxo de controle ou grafo de programa .

Segundo Vincenzi(2004), uma possível estratégia de teste é utilizar primeiramente os critérios funcionais e, posteriormente, utilizar os critérios estruturais para avaliar qual a cobertura obtida pelo conjunto de casos de teste funcional. Não se obtendo um conjunto de casos de teste adequado, esse deve ser complementado de modo a tornar-se adequado aos critérios estruturais. Isso significa que uma técnica complementa a outra, uma técnica sozinha nao é capaz de identificar todos os defeitos possíveis.

Como havia sido dito, o teste estrutural utiliza grafo de fluxo de controle para representação. Este grafo nada mais é senão um grafo que possui apenas um único nó de entrada e um único nó de saída. Cada vértice deste grafo representa um bloco não divisível de comandos e cada aresta é um possível desvio de um bloco para outro . Um bloco de comandos é a seqüência de um ou mais comandos com a propriedade de que, sempre que o primeiro comando do bloco é executado, todos os demais comandos do bloco também são executados e não existem desvios para o meio do bloco (SPOTO, 2000).

Figura 2: Exemplo de Grafo de fluxo de controle.



Na Figura 2 é mostrado um grafo de um programa (método) qualquer, onde os circulos representam os blocos, as setas são os arcos (arestas, que representam o fluxo de controle entre os nós). Todo grafo possui um nó de entrada e um nó de Saída.

Exemplos de critérios estruturais mencionados por Vicenzi (2004): Critérios Baseados na Complexidade, Critérios Baseados em Fluxo de Controle e Critérios Baseados em Fluxo de Dados.

2.3.1Critérios Baseados na Complexidade

Os critérios baseados na complexidade utilizam informações sobre a complexidade do programa para derivar os requisitos de teste (CYRILLO, 2004).

Um exemplo é o critério de McCabe. Este critério requer a execução de um conjunto de caminhos linearmente independentes do grafo de programa e utiliza a complexidade ciclomática para derivar os requisitos de teste.

Segundo Pressman(2000), a complexidade cicomatica é "[...] uma métrica de software que proporciona uma medida quantitativa de complexidade lógica de um programa[...]." Ou seja, o valor calculado da complexidade ciclomática define o número de caminhos independentes do conjunto básico do programa. Assim, ela oferece um limite máximo para o número de testes que devem ser realizados para garantir que todas as instruções sejam executadas pelo menos uma vez.

2.3.2 Critérios Baseados em Fluxo de Controle

Segundo Nardi (2006), os critérios baseados em fluxo de controle tem como foco os comandos do código. Ele define fluxo como sendo "[...] o curso que será tomado entre um comando e outro durante uma execução."

De acordo com Vicenzi(2004), alguns dos critérios de testes estruturais baseados em fluxo de controle existentes são:

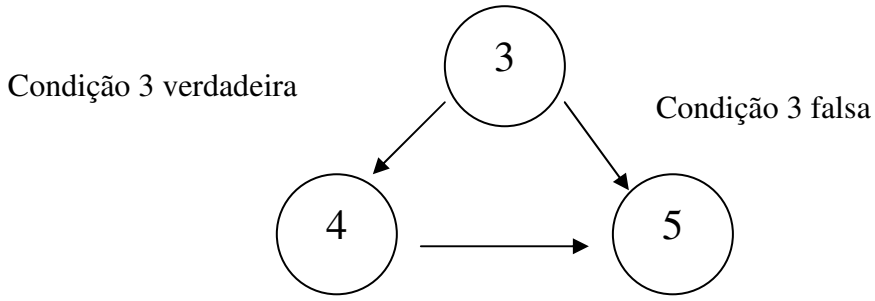
Todos nós: Segundo Vicenzi (2004), este tipo de teste exige que a execução do programa passe por cada vértice pelo menos uma vez no grafo de fluxo de controle.

Todos Caminhos: Requer que todos os caminhos possíveis sejam executados. Segundo Vincenzi(2004), geralmente é impraticável em apenas uma execução, pois pode haver casos em que o código possua um comando if e este irá se decidir por um dos caminhos apenas. Para que o outro caminho seja exercitado, deve-se fazer mais uma execução, utilizando valores diferentes do primeiro.

Todos arcos: De acordo com Vicenzi (2004), este tipo de critério exige que cada aresta do grafo seja exercitada ao menos uma vez.

No critério todos-nós é requerido que todos os comandos do método sejam executados pelo menos uma vez, algo que as vezes fica dificil de cumprir, pois há condições em certos programas que faz com que o fluxo se divida em dois caminhos possiveis. Isso acontece com estruturas if (se). Caso a condição seja verdadeira, um nó será executado, caso seja falsa, outro será executado, como é mostrado na Figura 3.

Figura 3 : Exemplo de grafo de programa com a confição If (se)

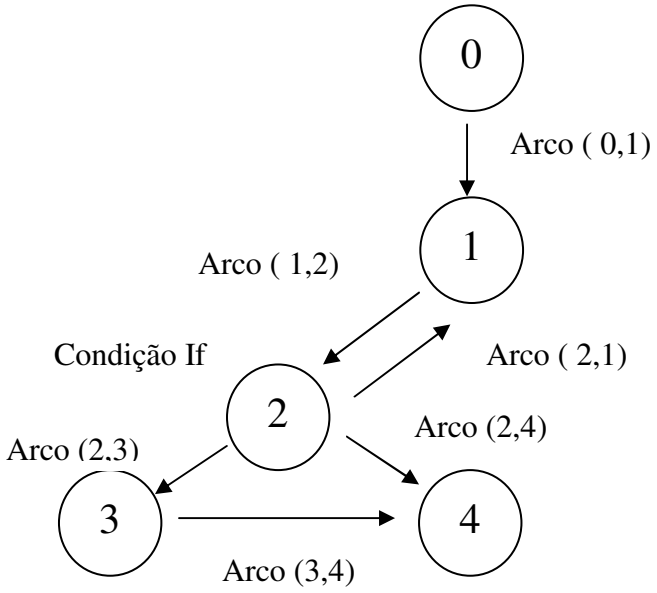


Em uma primeira execução, se a condição 3 for verdadeira, serão executados os nós <3 , 4 , 5>, ou, se a condição for falsa, será executado os nós <3, 5>. Portanto, apenas se a condição for verdadeira todos os nós serão executados. Mas se a condição for falsa, o nó não será executado. Para este caso deve-se desenvolver casos de teste que satisfaçam as duas condições o que não será possivel em apenas uma execução (NARDI, 2006).

Utilizando este critério seria possivel descobrir defeitos ocasionados por loops infinitos, pois como as entradas são feitas aleatoriamente é provável que não se digite uma entrada que satisfaça a condição de entrada do laço (NARDI, 2006).

Já com o critério todos os arcos, que requer que todos os arcos (arestas) sejam executadas, pode acontecer casos em que todos os nós foram executados porém nem todas as arestas foram, conforme será mostrado na Figura 4.

Figura 4: Exemplo de grafo utilizando o critério todos os arcos.



No caso de um código com o grafo da Figura 4 , utilizando o critério todos nós, poderíamos exercitar o caminho <0,1,2,3,4> e os arcos (0,1),(1,2),(2,3),(3,4). Veja que o arco (2,3), não foi exercitado porém todos os nós foram. Pode ser que a condição contida no bloco 2 seja um **if** cujas entradas foram suficientes apenas para seguir até o próximo bloco (3) e não satisfez a condição que o levasse ao bloco 4 diretamente. Seria necessário mais uma execução com entradas que satisfizessem as condições.

O critério todos-caminhos requer que todos os caminho sejam exercitados. Este critério pode ser impraticável de acordo com Nardi (2006), principalmente se ocorrer laços. Na Figura 5 é ilustrado melhor esta situação.

Na Figura 5 é mostrado o código de um programa qualquer com as condições apresentadas. No nó 0 é feita a definição das variáveis a e b, ou seja, estas variáveis receberão um valor neste bloco. Neste caso pode-se definir os seguintes caminhos:

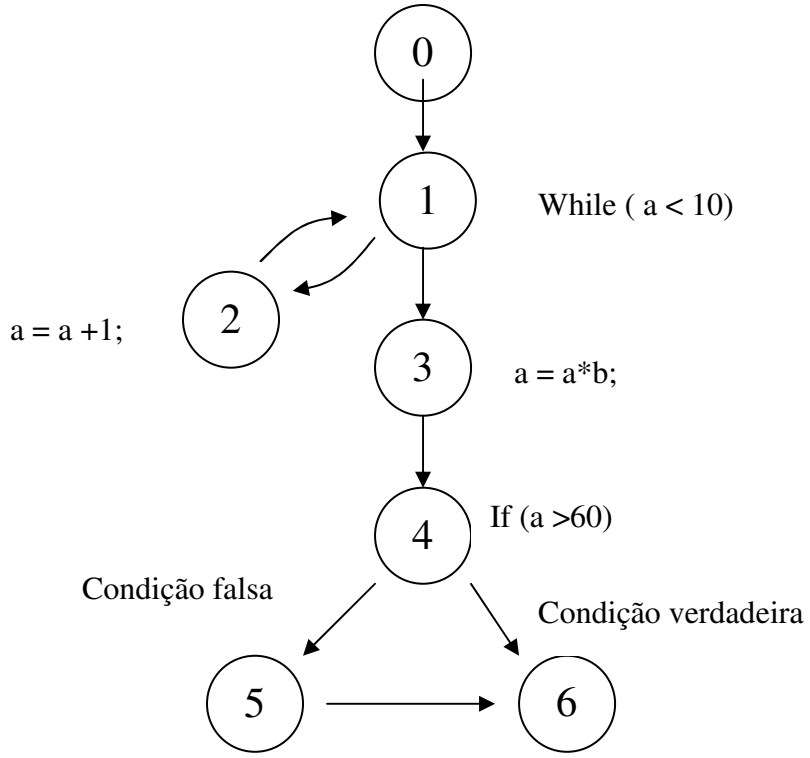
<0,1,3,4,6>

<0,1,2,1,2,1,2,1,2,1,3,4,5,6>

<0,1,2,1,2,1,2,1,2,1,3,4,6>

<0,1,3,4,5,6>

Figura 5: Exemplo de grafo de programa para utilização de critério todos-caminhos.



As possibilidades são várias. Para tentar cobrir todos os testes, deve-se inserir dados de entrada que satisfaçam estas condições. Por exemplo, caso as definições de a = 7 e b = 5, pode-se derivar o seguinte caminho:

< 0,1,2,1,2,1,2,1,3,4,5,6>.

Se a definição de b = 7, o caminho já seria < 0,1,2,1,2,1,2,1,3,4,6>.

De acordo com Nardi(2006) pode haver a ocorrência de caminhos infactíveis, que são aqueles que num primeiro momento pode se pensar em executá-lo, mas devido as condições do próprio código, seria impossível de se exercitar. Neste caso, o caminho <1,2,1> poderia nunca ser executável se o valor de a for maior do que 10.

Outro ponto a ser observado é a complexidade do teste. Se um laço existente em um código pode ser percorrido de 1 a 2000 vezes, haverá 2000 caminhos possíveis. O mesmo acontece com um comando case com 10 valores possíveis. Devido a quantidade de casos de testes que devem ser exercitados para satisfazer todas as condições, adotar este critério pode ser inviável.

2.3.3 Critérios Baseado em Fluxo de Dados

Como próprio nome já diz, utiliza informações do fluxo de dados do programa para derivar os requisitos de teste. Este tipo de critério requer que sejam testadas as interações que envolvem definições de variáveis e referências a essas definições (RAPPS, 1982 apud VICENZI, 2004). Exemplos: critério de Rapps e Weyuker (1982).

Para que se possa trabalhar com este tipo de critério, é preciso adicionar ao grafo de programa a informação sobre fluxo de dados, surgindo assim o Grafo Def-uso, que foi definido por Rapps e Weyuker.

> "Neste grafo são exploradas as associações entre a definição e o uso das variáveis determinando os caminhos a serem exercitados. Quando a variável é usada em uma computação, diz-se que seu uso é computacional (c-uso); quando usada em uma condição, seu uso é predicativo (p-uso). Os critérios desta classe são: Todas-Definições (all-defs), Todos-Usos (all-uses), Todos-Du-Caminhos (all-du-paths); Todos-P-Usos (all-p-uses), Todos-P-Usos/Alguns-C-Usos (all-p-uses/some-c-uses) e Todos-CUsos/ Alguns-P-Usos (all-c-uses/some-p-uses)."(Vicenzi, 2004)

Segundo Pressman(2000), este método seleciona caminhos de teste de um programa de acordo com as localizações das definições das variáveis (atribuição de valor às variáveis) e o usos de variável no programa.

Segundo Spoto(2000), a análise do fluxo de dados baseia nas ocôrrencias de variáveis de um programa de aplicação.

Variáveis de programas: são as variáveis do programa hospedeiro.

Variáveis de host: estas variáveis estabelecem os fluxos de dados entre a base de dados e o programa. Estas variáveis são fundamentais para estabelecer a comunicação entre as linguagens hospedeira e SQL.

Variáveis de tabela: são processadas apenas na linguagem SQL. Podem ser do tipo básico, que são variáveis persistentes que compõe a base de dados ou do tipo visão, que são tabelas virtuais derivadas de tabela básica e são tratadas como variáveis locais.

De acordo com Rapps e Weyuker (1985), define-se como c-Uso, ou uso computacional, quando a variável é usada na avaliação de uma expressão.

2.4 Critério de Teste Estrutural Orientados a Objetos

Vicenzi (2004) afirma que diversos autores consideram que estratégias de teste tradicionais não são eficazes em detectar defeitos em programas OO e a grande maioria dos trabalhos que já foram desenvolvidos focam no teste baseados em estados. Porém apenas este teste não é o suficiente para detectar todos os tipos de defeito em programas OO. De acordo com ele, são necessários outros critérios de teste, tais como critérios de fluxo de controle, critérios de fluxo de dados e critérios de fluxo de objetos para uma máxima detecção de defeitos.

Se considerarmos o teste de componentes, vamos observar que a grande maioria dos critérios de teste atualmente são critérios de teste funcional. Este tipo de critério não garante que partes importantes ou criticas do código tenham sido cobertas pelo conjunto de teste (VICENZI, 2004).

Todas essas considerações motivaram o estudo da aplicação de critérios de teste estruturais no teste de programas OO.

2.5 Critério de Teste Estrutural de ABDR

Esta seção contém uma breve introdução das principais características e conceitos básicos do modelo relacional que serão utilizados no teste estrutural de programas de Aplicação de Banco de Dados Relacional (ABDR). Também serão apresentados conceitos do critério de teste estrutural utilizado neste trabalho.

2.5.1 Conceitos de Banco de Dados Relacional (BDR)

O Modelo de Dados Relacional foi introduzido por Codd em 1970. De acordo com Spoto (2000), o modelo é baseado em uma estrutura de dados simples e uniforme chamada de "relação" com fundamentação teórica apoiada na álgebra relacional, ou seja, conjunto de operações para manipular as relações e especificar as consultas.

Como Spoto (2000) descreve, o modelo relacional representa uma base de dados como uma coleção de relações. Cada relação pode ser considerada como uma tabela. Cada linha da tabela representa uma variedade de valores. Estes valores podem ser fatos que represente uma entidade do mundo real. O nome da tabela ajuda a identificar os valores de cada linha. O nome de cada coluna ajuda a identificar os valores de cada coluna.

Uma linha é denominada de tupla e o título de cada coluna é um atributo e assim a tabela pode representar uma relação. Os tipos de dados identificam os tipos de valores de cada coluna e que são chamados de domínios dos atributos (SPOTO, 2000).

Um esquema de relação R, denotado por $R(A_1, A_2, ..., A_n)$, é formado pelo nome da relação R e um conjunto de atributos $A_1, A_2, ..., A_n$. Cada atributo A_i é o nome de um papel exercido por algum domínio D_i no esquema relacional R. D_i é chamado o domínio de A_i e é denotado por $dom(A_i)$ (SPOTO,2000).

Spoto (2000) ainda afirma que cada esquema de relação ou tabela possui seus atributos e as respectivas restrições, de modo a estabelecer as regras de integridade entre as relações.

2.5.2 Conceitos de Aplicação de Banco de Dados Relacional (ABDR)

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

É composta por um ou vários programas que suportam comandos da linguagem SQL. A **SQL** usa os termos tabela, linha e coluna para representar uma relação, tupla e atributo, respectivamente.

De acordo com Spoto(2000),os comandos da SQL são classificados em dois tipos: declarativos e executáveis. Os declarativos são aqueles utilizados para declaração de objetos, variáveis de controle e áreas de ligação. Os executáveis são divididos em dois blocos: Comandos de Definição e Comandos de Manipulação de Dados, como pode ser visto na tabela abaixo:

Tabela 02: Comandos executáveis

Comandos de Definição	Comandos de Manipulação de Dados
ALTER, CREATE, DROP, RENAME	DELETE, INSERT, UPDATE
	CLOSE, FETCH, OPEN, SELECT
CONNECT, GRANT, LOCK TABLE, REVOKE	COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION
	DESCRIBE, EXECUTE, PREPARE

2.5.3 Teste estrutural de ABDR

O teste estrutural de programas de ABDR com SQL embutida apresenta algumas características similares ao do teste de programas convencionais. De acordo com Spoto(2000),

“[...] técnicas convencionais de teste podem ser aplicadas em programas de aplicação a partir de algumas adaptações necessárias, tendo em vista a presença de comandos SQL e variáveis tabelas e variáveis host, não existentes em programas convencionais [...].”

Devido à existência dos comandos de SQL em programas hospedeiros de aplicações de Banco de Dados relacional, a definição de grafo de programa foi alterada para acomodar os comandos executáveis da SQL em nós especiais, um em cada nó.

Na representação gráfica, os comandos da linguagem hospedeira e os comandos declarativos da SQL podem ser acomodados em blocos de comandos (SPOTO, 2000). Os comandos executáveis da Linguagem SQL (INSERT, DELETE, UPDATE, SELECT, COMMIT, ROLLBACK, entre outros) são acomodados isoladamente.

Adota-se nós circulares para representar os blocos de comandos (linguagem hospedeira e comandos declarativos da SQL) e nós retangulares para representar os comandos executáveis da SQL. As setas, denominadas de arcos, representam possíveis transferências de controle entre os nós (SPOTO, 2000).

A abordagem de teste estrutural para programas de ABDR, segundo Spoto (2000) considera dois tipos de fluxos de dados: fluxo intra-modular, que é usado para o teste estrutural de cada Módulo de Programa da aplicação e o fluxo intra-modular, usado no teste de unidade e no teste de integração intra-modular; este último é efetuado para a integração das unidades de programa.

O fluxo inter-modular é utilizado para o teste estrutural considerando o fluxo de dados entre os Módulos de Programas que compõem a ABDR. Esse fluxo de dados envolve as variáveis tabelas da base de dados. O teste baseado neste tipo de fluxo de dados foi denominado de teste de integração inter-modular, efetuado para integrar os diferentes Módulos de Programas existentes na aplicação (SPOTO,2000). Spoto (2000) denomina de Módulo um Programa que compõem a Aplicação. Neste contexto iremos denominar de intra-classe por considerar integração de uma mesma classe e inter-classe quando integrar métodos de classes distintas.

A seguir são apresentados os conceitos básicos do teste de integração baseado nas chamadas de procedimentos em programas de aplicação.

O grafo de chamada é um multigrafo, onde podem ocorrer mais de uma chamada entre duas unidades de programa. A integração entre duas unidades de programa é feito de dois a dois e conseqüentemente os requisitos de teste são derivados para cada par (SPOTO, 2000).

Os critérios de integração são derivados de teste de unidade e visa associar pontos de definição e uso para variáveis globais ou passadas por paramentos. São consideradas variáveis de programa e variáveis host.

Na seção a seguir, são apresentados os conceitos de teste de integração para aplicações OO.

2.6 Critérios de Teste de Aplicação OO para Abdr

Uma abordagem de teste de integração entre os métodos e classes de um programa orientado a objetos foi apresentada por Harrold e Rothermel. Essa abordagem testa os métodos e classes segundo três níveis: i) *teste intra-método*: testa os métodos (operações) individualmente, que é equivalente ao teste de unidade; ii) *teste inter-mét*odos: testa um método público junto com outro método em sua classe que o chama direta ou indiretamente (equivale ao teste de integração interprocedimental); iii) *teste intra-classe*: testa as interações de métodos públicos quando são chamados em várias seqüências (SPOTO,2000).

Neste capitulo são apresentados os conceitos do teste intra-classe, que é o utilizado neste trabalho.

2.6.1 Intra-classe

O teste intra classe envolve o teste de unidade (fluxo de dados persistente que são relacionados em um mesmo método) e teste de integração entre métodos que associam uma mesma tabela (definição persistente) com uso em outro método (da mesma tabela).

A associação que será estudada aqui envolve o critério todos t-usos (intra ou inter) métodos de uma mesma classe.

O t-uso é uma associação definição persistente de t (tabela) com o uso da mesma tabela t. Definição persistente é o par (INSERT, DELETE OU UPDATE com o comando COMMIT). <$NO_{DELETE,\ UPDATE,\ INSERT}$, NO_{COMMIT}> já o uso de t será o arco de saída dos comandos (INSERT, DELETE, UPDATE OU SELECT) ($NO_{INSERT,\ DELETE,\ UPDATE\ OU\ SELECT}$, NOSEGUINTE (SPOTO, 2000).

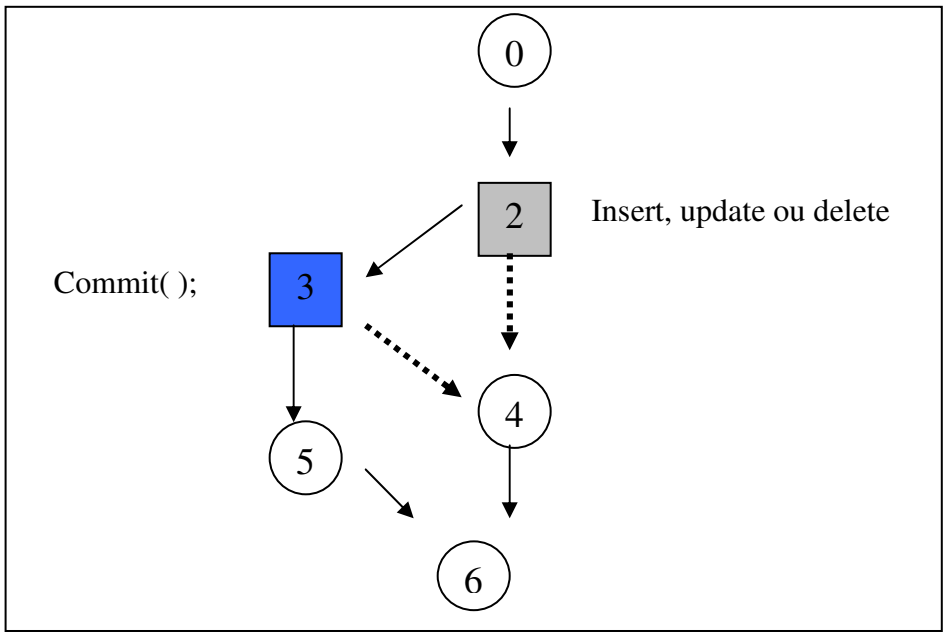
Considere o grafo abaixo como sendo de um método de inserção, alteração ou deleção de uma classe chamada Cadastro que realiza o gerenciamento de funcionarios.

No teste intra-classe pode-se ter gerar duas etapas de testes: teste de unidade e teste de integração.

2.6.1.1 Teste de Unidade

É a etapa aplicada em cada método de uma determinada Classe isoladamente. O teste de unidade inicia ainda na etapa de desenvolvimento, antes que todas as unidades do sistema seja integrada.

Figura 6: Exemplo de grafo de um programa Cadastro de funcionarios

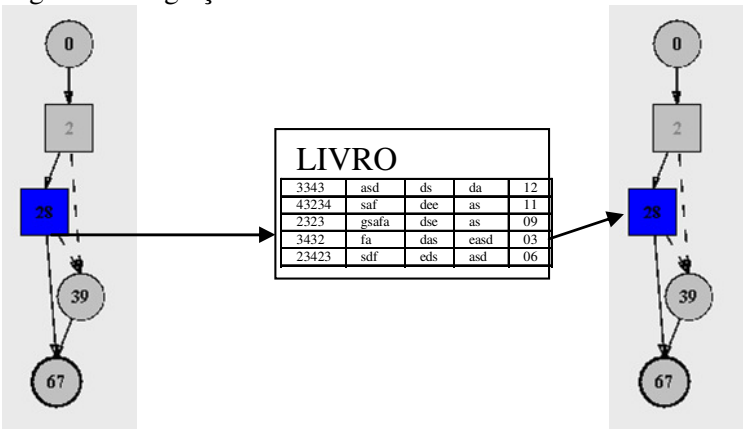


Na Figura 6, os nós quadrados representam os comandos do banco. Os comando de insert, delete e update estao no nó 2 e o commit está no nó 3. Então, um possivel elemento requirido de teste é <funcionario <2,3>, (2,3)> caso a inserção, alteração, deleção seja realizada com sucesso no banco ou < funcionario <2,3>, (2,4)> caso ocorra algum erro na inserção, alteração, deleção.

2.6.1.2 Integração entre os métodos

O Teste de integração pode ser realizado integrando as unidades aos pares. No caso de uma classe o teste de integração inter-métodos (de uma mesma classe) realiza os fluxos de informação existentes entre os métodos. Na Figura 7 é mostrada uma representação entre duas unidades sendo que a primeira define um valor na tabela e a segunda usa um valor da tabela. Desta forma é aplicado o critério todos t-usos-intra (SPOTO, 2000), sendo este critério utilizado neste trabalho.

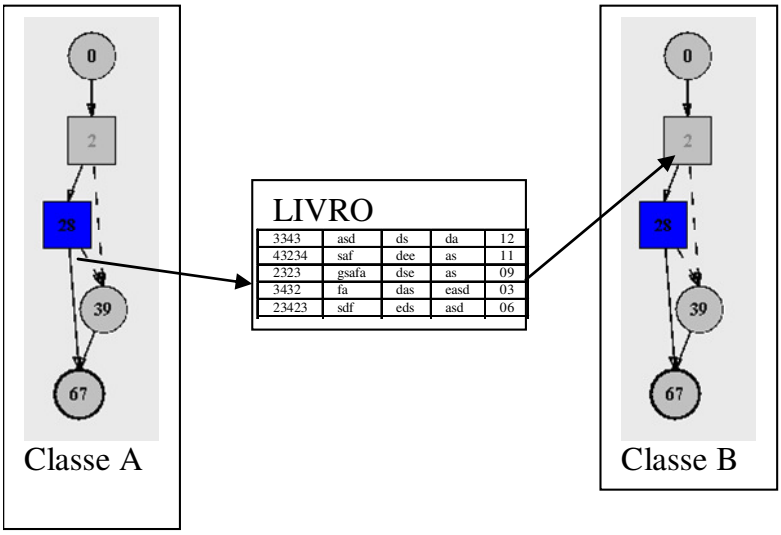
Figura 7: Integração entre dois métodos de uma mesma classe



2.6.2 Inter-classe.

Os testes inter-classe tem o mesmo objetivo do que o teste intra-classe, porém neste tipo de teste não é necessário que esteja na mesma classe, pode estar em classes diferentes. Neste caso, é necessário indicar a classe onde está a definição persistente e a classe onde está o uso (MOLINA, 2006). No exemplo da Figura 8 é mostrada a integração entre dois métodos de classes distintas (inter-classe).

Figura 8: Integração entre dois métodos de classes distintas



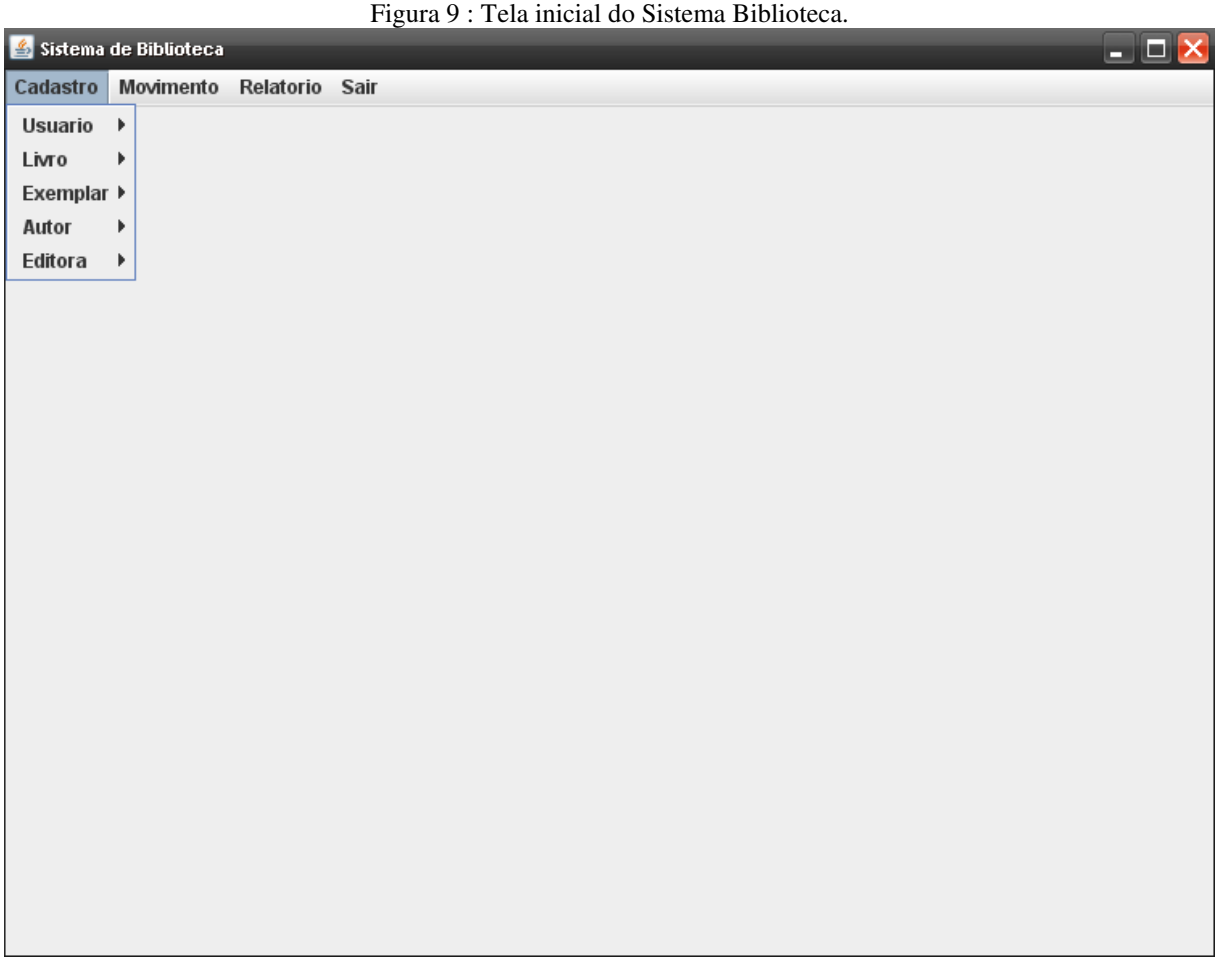
CAPITULO 3 - ESTUDO DE CASOS

Neste capitulo é apresentado o sistema utlizado para a realização dos teste, a geração dos casos de testes de acordo com os elementos requeridos e os resultados obtidos.

3.1 Sistema Biblioteca.

Para a realização dos testes, foi utilizado uma aplicação que faz o gerenciamento de uma Biblioteca. O Sistema Biblioteca é um software para gerenciamento de usuários, livros, autores, editores, emprestimos e devoluções de livros. Na Figura 9 é apresentada a tela inicial do programa.

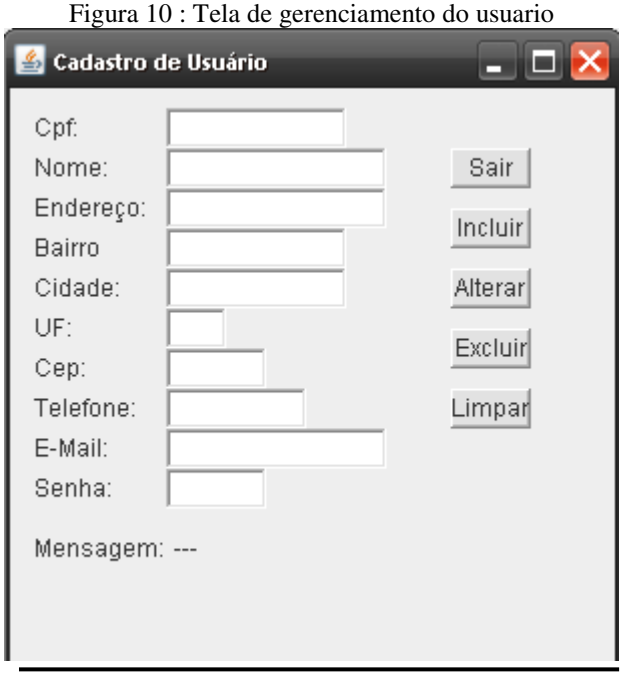
Figura 9 : Tela inicial do Sistema Biblioteca.



Este sistema atualmente só tem implementado a parte de gerenciamento de Usuário. Por isso que só será realizado testes intra-classe neste programa no momento. No Anexo A é apresentado o código desta aplicação

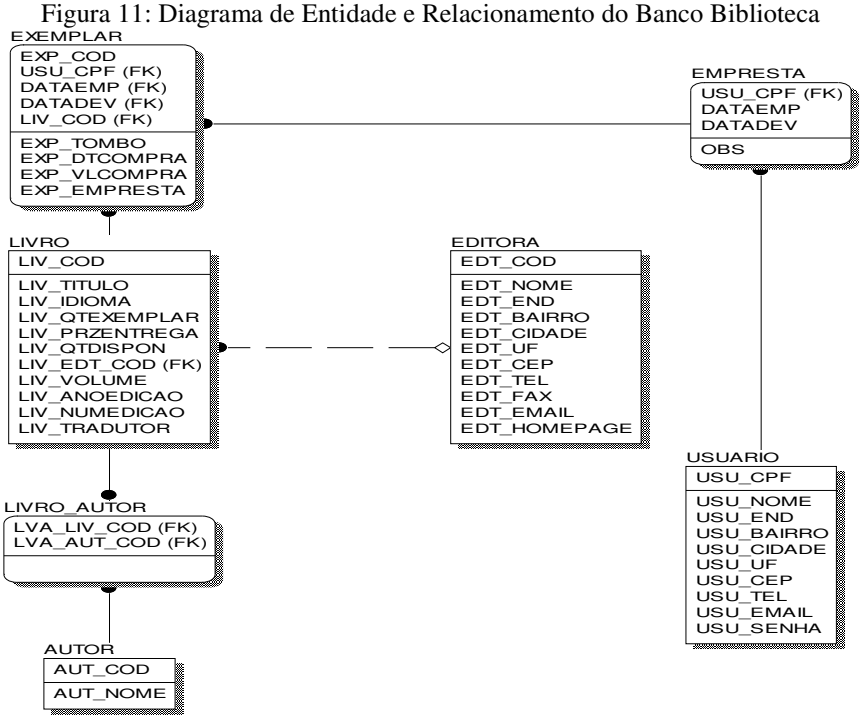
O gerenciamento do usuario é um módulo que permite a inclusão de um usuário, alteração de seus dados e sua exclusão , como mostrado na Figura 10. Todas as buscas para alteração e exclusão são feitas a partir do CPF do usuário.

Figura 10 : Tela de gerenciamento do usuario



Na Figura 11, será apresentao o Diagrama Entidade Relacionamento (DER) das tabelas deste sistema:

Figura 11: Diagrama de Entidade e Relacionamento do Banco Biblioteca

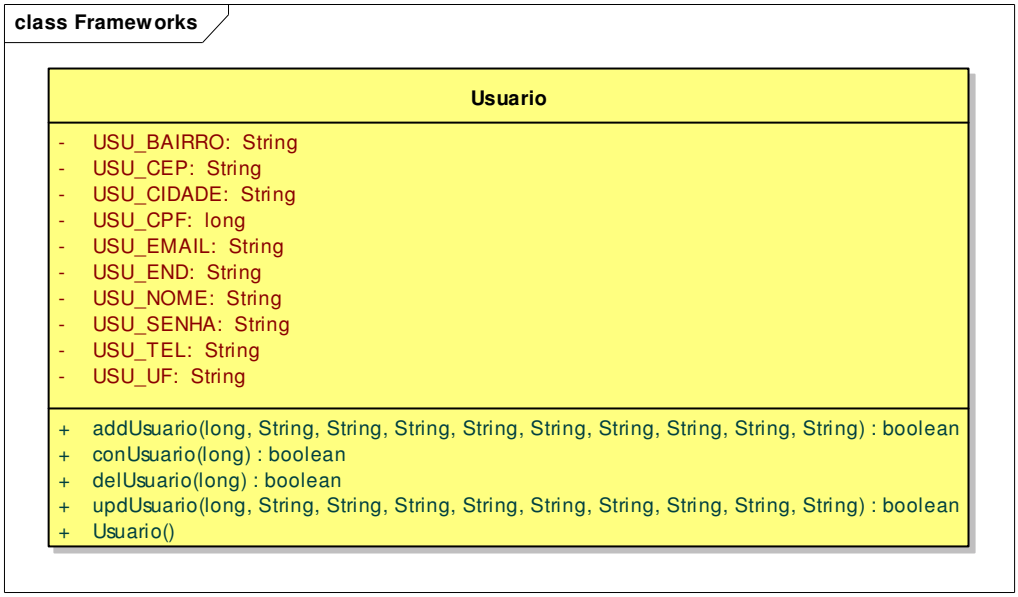


Como dito anteriormente, todas as buscas para usuário são feitas através do CPF. O campo CPF é um campo number, de tamanho 11. Os scripts de criação das tabelas e o códigos das classes mencionadas poderão ser encontrados no Anexo A e B, respectivamente, no final deste trabalho.

3.2 Apresentação das classes.

Neste capitulo serão apresentadas as classes principais utilizadas para a realização deste trabalho. No anexo C pode ser encontrado o diagrama geral do sistema.

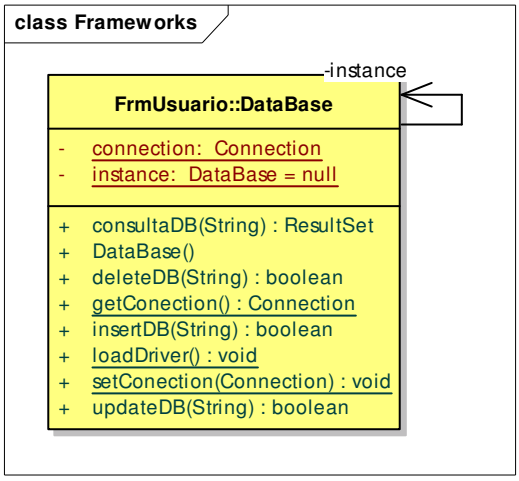
Figura 12: Classe Usuário



Na Figura 12, é apresentada a classe Usuario. Esta classe recebe os dados de inserção, alteração e exclusao do usuário que sao digitados na aplicação.

Os métodos desta classe monta a string sql específica para cada operação que cada método executa e então passa essa string como parâmetro para a classe Database que se encontra representada na Figura 13.

Figura 13: Classe Database



A Classe Database mostrada na Figura 13 faz a conexão de todos os acessos ao banco de dados. Os métodos utilizados para a realização dos testes são estão nesta classe. São eles: insertDB(), updateDB() e deleteDB(), contidos na classe Database, conforme mostrado na Figura 12.

O método insertDB( ) realiza a inserção dos dados no banco de dados. Na classe Usuario, existe um método chamado addUsuario conforme mostrado na Figura 12, que recebe os dados do usuário inseridos na aplicação, monta o comando de inserção em uma string chamada sql e passa como parâmetro para a classe insertDB(String sql) que após a execução do commit( ), insere os valores no banco de dados.

Já o método updateDB(String sql) faz a atualização dos dados no banco de dados. Ela tem o funcionamento parecido com o método insertDB( ). Quem recebe os dados para alteração é o método updUsuario( ) da classe Usuário. Mas antes devemos falar do método conUsuario( ), da classe Usuário.

O método conUsuario( ) é um método do tipo verdadeiro ou falso, que serve para fazer consultas no banco de dados. Ele possui em seu código o comando select que faz a busca os dados e retorna verdadeiro caso o dado seja encontrado ou falso caso não seja. Este método aparece nos métodos updateDB( ) para verificar se dado que se deseja alterar existe e no método deleteDB( ) para verificar se dado que se deseja excluir existe.

Voltando ao método updateDB( ), após a busca ter retornado que o dados existem para o método updUsuario(), da classe Usuário , este método monta a string sql, passa como parâmetro para o método updateDB( ), e este, através do comando commit( ), realiza a alteração dos dados no banco.

E por fim, o método deleteDB( ), que tem seu funcionamento identico ao updateDB(), inclusive sua interação com o método delUsuario da classe usuario, que funciona do mesmo modo que o método updUsuario descrito acima.

Nas próximas seções serão apresentadas as estrategias de testes e os casos de testes.

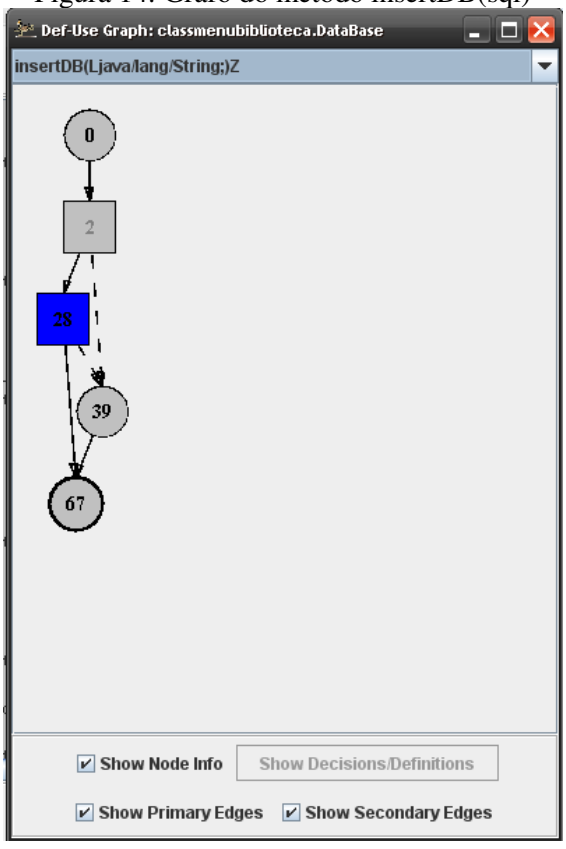
3.3 Casos de teste

Como visto anteriormente, o critério todos-t-uso intra-classe utilizado neste trabalho, requer uma associação de fluxo definição-persistente e t-uso, de uma mesma tupla.

Os casos de teste devem ser elaborados para poder satisfazer as condições descritas acima. Para isso, deve-se desenvolver dados de entrada que satisfaçam os elementos requeridos.

A seguir, vamos apresentar o grafo do método insertDB( ) gerado pela ferramenta Jabuti para exemplificar a montagem dos casos de teste de acordo com os elementos requeridos.

Figura 14: Grafo do método insertDB(sql)



Na Figura 14 é mostrado o grafo do método insertDB(sql), gerado pela ferramenta Jabuti. Segue abaixo a apresentação do código deste método para um melhor entendimento.

Código do metodo insertDB()

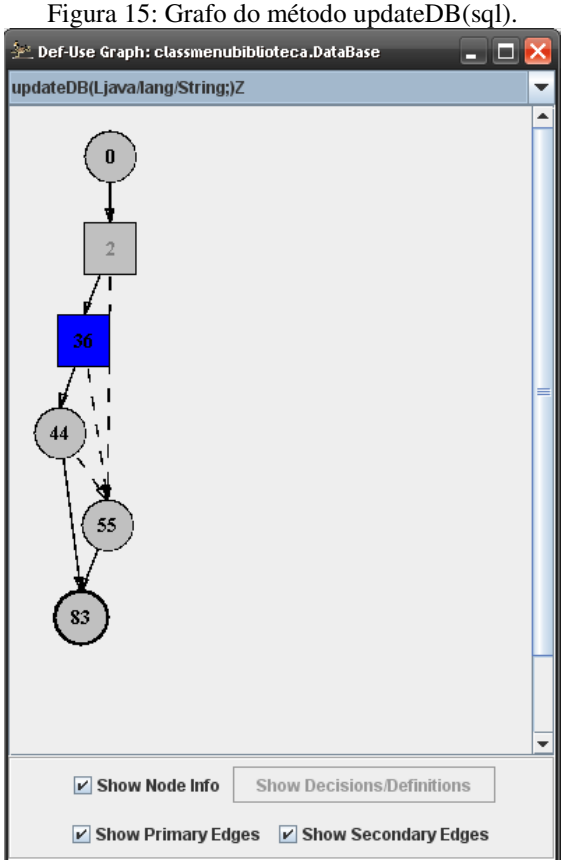
```
public boolean insertDB(String sql)
{
     boolean okInsert = false;
     Statement stm;
     try
     {
            //mostrando a sql no debug
            System.out.println(sql);
            Connection conexao=null;
            //criando o objeto Statement stm
            stm = getConection().createStatement();
            //executando a sql
            okInsert = stm.execute(sql);
            conexao.commit();
     }
     catch (SQLException sqlEx)
     {
            System.err.println("Erro InsertDB--> "+sqlEx);
     }
     //retornando se ok sql
     return okInsert;
}
```

Como pode ser observado acima, a linha do código destacada em cinza corresponde ao nó 2 exibido no grafo. É neste nó onde estão os comando sql executados, conforme também é mostrado no código. A montagem do comando sql é feito em outra classe (Usuário) e passado como parâmetro para o método insertDB(sql).

Logo em seguida, vemos destacado em azul o comando commit( ), representado pelo nó 28, no grafo em azul. É neste nó que ocorre a definição persistente requerida pelo critério todos-t-uso. Caso aconteça algum erro na inserção dos dados, o commit entao não será executado, sendo executado a parte do código representada pelo grafo de numero 39, o que no código é representado pelas linhas destacadas em verde.

Um t-uso acontece quando um dos comandos Insert, delet, update ou select são executados. Por isso, um t-uso para esta mesma situação também pode acontecer nos nós <2,28> neste mesmo método, ou em um outro método como será mostrado abaixo:

Figura 15: Grafo do método updateDB(sql).



Na Figura 15 é mostrado o grafo do método updateDB( ) gerado pela ferramenta Jabuti. A seguir, será mostrado o código deste método:

Código do metodo updateDB( )

```
public boolean updateDB(String sql)
     {
          boolean okUpdate = false;
          Statement stm;
          try
          {
               //mostrando a sql no debug
               System.out.println(sql);
               //criando o objeto Statement stm
```

```
                    stm = getConection().createStatement();
                    System.out.println("Create Statement update");
                    //executando a sql
                    okUpdate = stm.execute(sql);
                    Connection conexao=null;
                    conexao.commit();
                    System.out.println("Dados alterados");
                    System.out.println("Passou execute update");
            }
            catch (SQLException sqlEx)
            {
                  System.err.println("Erro updateDB--> "+sqlEx);
            }

      //retornando se ok sql
      return okUpdate;
     }
```

Como já foi comentado na seção anterior, o método updateDB( ) realiza a alteração dos dados no banco. De acordo com o grafo acima, o nó 2 representa as linhas do código descrito acima que estão grifados em cinza. Neste grafo, um t-uso para a situação descrita anteriormente acontece no nó 2, pois neste comando SQL que foi passado como parâmetro possui um comando **update** definido na classe usuário pelo método updUsuario. O nó 36 representa a linha do comando **commit**, grifado em azul no código, usado para gravar no banco as alterações.

Os grafos dos outros métodos utilizados, podem ser encontrados no Anexo D, no final deste trabalho.

Um elemento requerido para o critério todos-t-uso intra-classe segue o seguinte padrão:

<TABELA, métodoDef <NÓ_DEF, NÓ_COMMIT>, métodoUso(NÓ_USO, NÓ_SEGUINTE)>

Assim, pode-se desenvolver os elementos requeridos para o grafo da Figura 15, que para este caso poderia ser:

<USUARIO, insertDB<2,28>, updateDB<2,36>,

E um caso de teste para este elemento poderia ser inserir um usuário com um CPF 234354577565 e alterar os dados deste mesmo usuario, pois o critério requer que os teste sejam com a mesma tupla. Entao a execução dos casos de teste é feita e é observado se foi possivel exercitar o elemento requerido ou não, se foi detectado erros e se o comportamento foi o esperado.

Após o entendimentos destes conceitos, foram elaborados os elementos requeridos do critérios todos-t-usos-intra-classe que serão apresentados a seguir.

1 <USUARIO, insertdb, <2,28>,insertdb, (2,28)>.
2 <USUARIO,insertdb,<2,28>,insertdb,(2,39) >
3 <USUARIO,insertdb,<2,28>updatedb,(2,36) >
4 <USUARIO,insertdb,<2,28>, deletedb,(2,28) >
5 <USUARIO,insertdb,<2,28>,insertdb,(2,39) >
6 <USUARIO,updatedb,<2,36>,updatedb,(2,55) >
7 <USUARIO,deletedb,<2,28>,deletedb,(2,39) >
8 <USUARIO,insertdb,<2,28>,insertdb,(2,39) >
9 <USUARIO,insertdb,<2,28>updatedb,(2,55) >
10 <USUARIO,updatedb,<2,36>,insertdb,(2,28) >
11 <USUÁRIO, deletedb,<2,28>, insertdb,(2,28) >

Para cada um dos elementos requeridos, foram definidos os seguintes casos de teste:

1 Inserir usuario com cpf = 76565454344.
Inserir novamente este mesmo usuário.
2 Inserir usuario com cpf = 88776453324 .
Alterar a cidade deste usuário para Marilia.
3 Inserir usuario com cpf = 34234254334.
Excluir este mesmo usuário.
4 Inserir usuario com cpf = ‘brasilia’
5 Alterar dados de usuario com cpf = 99999999988
6 Excluir usuario com cpf = jkiu889
7 Excluir usuario com cpf = 76565454344.
Excluir novamente este usuário.

8	Inserir usuário com	cpf = 76565455121, email = jon@yahoo.com.br senha = 123c
	Inserir usuário com	cpf = 23244354312, email = jon@yahoo.com.br senha = jui889
9	Inserir usuário com	cpf = 87867756644, email = serra@yahoo.com.br senha = 456t
	Inserir usuário com	cpf = 56734213244, email = alves@yahoo.com.br senha = 456t
10	Alterar os dados do usuario com cpf = 87898943455	
	Inserir o mesmo usuário	
11	Excluir os dados do usuario com cpf = 87898943455	
	Inserir o mesmo usuário	

3.4 Execução dos casos de Teste

Para cada caso de teste descrito na seção anterior, foram definidas as Saídas esperadas. Pode-se dizer que foram necessárias duas execuções de entrada de dados para cada caso de teste para satisfazer os elementos requeridos, portanto tem-se duas Saídas esperadas e duas Saídas obtidas para cada caso, como são apresentadas a seguir:

Caso de teste 01:

Inserir usuario com cpf = 76565454344

Saída obtida: Dados inseridos com sucesso.

Saída esperada: Dados inseridos com sucesso. (Definição)

Inserir novamente usuario com cpf = 76565454344

Saída obtida:Erro. Usuario já cadastrado

Saída esperada: Usuario já cadastrado.

Status

Elemento requerido numero um não pode ser exercitado, ocorreu erro, mas erro já era esperado.

Caso de teste 02:

Inserir usuario com cpf = 88776453324.

Saída esperada: Dados inseridos com sucesso

Saída obtida: dados inseridos com sucesso

Alterar a cidade deste usuário para Marilia

Saída esperada: Dados alterados com sucesso

Saída obtida: Dados alterados com sucesso

Status

Exercitado.

Caso de teste 03:

Inserir usuario com cpf = 34234254334

Saída esperada: Dados inseridos com sucesso

Saída obtida: Dados inseridos com sucesso

Excluir este mesmo usuário

Saída: esperada Excluido com sucesso

Saída obtida: Excluido com sucesso

Status

Exercitado

Caso de teste 04:

Inserir usuario com cpf = Brasília

Saída esperada: Erro

Saída obtida: Erro. CPF inválido.

Status

Elemento requerido não pode ser exercitado, ocorreu erro, mas erro já era esperado.

Caso de teste 05:

Alterar dados de usuario com cpf = 99999999988

Saída esperada: Erro.

Saída obtida: Erro. CPF Invalido

Status

Elemento requerido não pode ser exercitado, ocorreu erro, mas erro já era esperado.

Caso de teste 06:

Excluir dados de usuario com cpf = jkiu889

Saída esperada: Erro.

Saída obtida: Erro. CPF Invalido

Status

Elemento requerido não pode ser exercitado, ocorreu erro, mas erro já era esperado.

Caso de teste 07:

Excluir usuario com cpf = 76565454344

Saída: esperada Excluido com sucesso

Saída obtida: Excluido com sucesso

Excluir este mesmo usuário

Saída: esperada: Erro, já foi excluido

Saída obtida: Excluido com sucesso

Status

Elemento requerido não pode ser exercitado, ocorreu erro.

Caso de teste 08:

Inserir usuário com cpf = 76565455121, email = jon@yahoo.com.br e senha = 123c

Saída esperada : Dados inseridos com sucesso

Saída obtida: Dados inseridos com sucesso

Inserir usuário com cpf = 23244354312, email = jon@yahoo.com.br e senha = jui889

Saída esperada: Erro, email já esta sendo utilizado

Saída obtida: Dados inseridos com sucesso

Status

Elemento requerido não pode ser exercitado, ocorreu erro.

Caso de teste 09:

Inserir usuário com cpf = 87867756644, email = serra@yahoo.com.br e senha = 456t

Saída esperada : Dados inseridos com sucesso

Saída obtida: Dados inseridos com sucesso

Inserir usuário com cpf = 56734213244, email = alves@yahoo.com.br e senha = 456t

Saída esperada: Erro, senha já esta sendo utilizado

Saída obtida: Dados inseridos com sucesso

Status

Elemento requerido não pode ser exercitado, ocorreu erro.

Caso de teste 10:

Alterar os dados do usuario com cpf = 87898943455

Saída esperada : Erro. CPF não existe

Saída obtida: CPF não existe

Inserir o mesmo usuário

Saída esperada: Dados inseridos com sucesso

Saída obtida: Dados inseridos com sucesso

Status

Exercitado

Caso de teste 11:

Excluir os dados do usuario com cpf = 87898943455

Saída esperada : Dados inseridos com sucesso

Saída obtida: Dados inseridos com sucesso

Inserir o mesmo usuário

Saída esperada: Erro, email já esta sendo utilizado

Saída obtida: Dados inseridos com sucesso

Status

Exercitado

Na seção seguinte será apresentado uma planilha com os resultados obtidos.

3.5 Resultados obtidos

Com os elementos requeridos definidos e o casos de teste desenvolvidos, foi feita a execução dos casos de teste, conforme foi descrito no capitulo anterior. Para apoiar esta atividade, foi elaborada uma planilha, que segue abaixo:

Tabela 3: Elementos Requeridos.

Elementos requiridos					
<nome tabela	*metodo Definição*	*DefiniçãoPersistente*	*metodo Uso*	*T-Uso*	**Status**
<USUARIO	insertdb	<2,28>	insertdb	(2,28)	exercitado
<USUARIO	insertdb	<2,28>	insertdb	(2,39)	exercitado *
<USUARIO	insertdb	<2,28>	insertdb	(2,28)	exercitado
<USUARIO	updatedb	<2,36>	updatedb	(2,36)	exercitado
<USUARIO	insertdb	<2,28>	insertdb	(2,28)	exercitado
<USUARIO	deletedb	<2,28>	deletedb	(2,28)	exercitado
<USUARIO	insertdb	<2,28>	insertdb	(2,39)	exercitado *
<USUARIO	updatedb	<2,36>	updatedb	(2,55)	exercitado *
<USUARIO	deletedb	<2,28>	deletedb	(2,39)	exercitado *
<USUARIO	deletedb	<2,28>	deletedb	(2,28)	exercitado
<USUARIO	deletedb	<2,28>	deletedb	(2,39)	aplicação exclui funcionario que não existe, pois não verifica se o cpf já foi deletado ou não
<USUARIO	insertdb	<2,28>	insertdb	(2,28)	exercitado
<USUARIO	insertdb	<2,28>	insertdb	(2,39)	aplicação deveria verificar se os logins (emails são iguais)
<USUARIO	insertdb	<2,28>	insertdb	(2,28)	exercitado
<USUARIO	updatedb	<2,36>	updatedb	(2,55)	aplicação deveria verificar se senhas são iguais e não verificou
<USUARIO	updatedb	<2,36>	updatedb	(2,55)	exercitado
<USUARIO	insertdb	<2,28>	insertdb	(2,28)	exercitado
<USUARIO	deletedb	<2,28>	deletedb	(2,39)	a aplicação não informa que o cpf não existe e exclui mesmo assim
<USUARIO	insertdb	<2,28>	insertdb	(2,28)	exercitado

*significa que o elemento requerido desta linha não é exercitavell porem o resultado esperado foi obtido.

Na Tabela 3 são mostrados os elementos requeridos desenvolvidos para atender ao critério todos-t-usos intra classe para facilitar a identificação e o desenvolvimento dos casos de teste, que serao mostrados a seguir. A última coluna, denominada Status, mostra quais os resultados que se chegaram com a execução dos testes. A partir desta planilha foram definidos os elementos requeridos do critério utilizado.

Tabela 4: Casos de teste

Casos de Teste				
Operação	*dados de entrada*	*outros dados de entrada necessarios*	*saida esperada*	*saida obtida*
insert	cpf = 76565454344		dados inseridos com sucesso	dados inseridos com sucesso
insert	cpf = 76565454344		erro. Cpf já cadastrado	Usuario já cadastrado
insert	cpf = 88776453324		dados inseridos com sucesso	dados inseridos com sucesso
update	cpf = 88776453324	cidade = marilia	dados alterados com sucesso	dados alterados com sucesso
insert	cpf = 34234254334		dados inseridos com sucesso	dados inseridos com sucesso
delete	cpf = 34234254334		dados deletados com sucesso	Excluido com sucesso
insert	cpf = brasilia		erro. Cpf invalido	erro.
update	cpf = 99999999988		erro. Cpf não cadastrado	usuario não cadastrado
delete	cpf = jkiu8899		erro. Cpf invalido	erro.
delete	cpf = 76565454344		dados deletados com sucesso	Excluido com sucesso
delete	cpf = 76565454344		erro. Cpf não existe(já foi deletado)	Excluido com sucesso
insert	cpf = 76565455121	email = jon@yahoo.com.br senha = 123c	dados inseridos com sucesso	dados inseridos com sucesso
insert	cpf = 23244354312	email = jon@yahoo.com.br senha = jui889	erro: email já existe	dados inseridos com sucesso
insert	cpf = 87867756644	email = serra@yahoo.com.br senha = 456t	dados inseridos com sucesso	dados inseridos com sucesso
update	cpf = 56734213244	email = alves@yahoo.com.br senha = 456t	erro: senha já existe	dados inseridos com sucesso
update	cpf = 87898943455		erro: cpf não existe	usuario não cadastrado
insert	cpf = 87898943455		dados inseridos com sucesso	dados inseridos com sucesso
delete	cpf = 76563435223		erro: cpf não existe	Excluido com sucesso
insert	cpf = 76563435223		dados inseridos com sucesso	dados inseridos com sucesso

Na Tabela 4 são mostrados os casos de testes definidos para satisfazer cada elemento requerido. A coluna “saída obtida” mostra o resultado obtido após a execução dos testes na aplicação. Esta coluna foi a base para a coluna Status da Tabela 3. Para alguns casos de testes foram necessário dados de entrada adcionais, como pode ser visto na Tabela 4.

CONCLUSÃO E TRABALHOS FUTUROS

O objetivo deste trabalho era saber se técnicas estruturais de teste de ABDR podiam ser aplicadas em sistemas orientados a objetos com banco de dados relacional. Para isso, foram discutidos neste trabalho os conceitos da linguagem orientada a objetos e seu impacto na estrategia de teste após estudos realizados por Vincenzi(2004). Vimos o quanto é importante o desenvolvimento de técnicas que garantam um teste de alta qualidade para sistemas orientados a objetos.

Foi estudado também os critérios estruturais existentes e as técnicas mais utilizadas, optando pelo teste de integração intra-classe e o critério todos-t-uso.

Foi feita então uma tabela com os elementos requeridos deste critério, afim de se montar casos de teste para exercitar estes elementos requeridos e verificar se a utilização deste critério é util na descoberta de erros em aplicações OO com banco de dados relacional.

De acordo com os resultados obtidos após a execução dos casos de teste, foi possivel observar que a maioria dos elementos requeridos foram exercitados com sucesso. Isto significa que o sistema correspondia ao esperado para uma boa implementação do projeto de banco de dados e da aplicação java.

Também foi percebido que alguns erros que não estavam previstos foram descobertos e também outros, mas que estavam previstos. Ou seja, o critério utilizado todos t-usos (no teste de integração intra-classe) foi capaz de encontrar erros na aplicação e no banco que estavam previstos e até alguns não previstos.

O que nos leva a concluir que a utilização deste critério é util para verificar o quanto um sistema de aplicação corresponde com as características de um banco de dados ao que se refere à sua implementação respeitando as exigências de integridades do Banco.

Neste trabalho não foi necessária a aplicação da técnica de teste todos-t-usos-inter-classe pois todos os acessos ao banco de dados são feitos através de uma mesma classe chamada Database, através dos métodos insertDB( ), updateDB( ), deleteDB( ) e consultaDB( ).

As técnicas de teste são complementares, sendo assim as técnicas de teste estrutural para BD aplica-se somente para a avaliação do fluxo de informação que está sendo gerado do programa para o Banco de dados e vice-versa. Outras técnicas deverão ser utilizadas para avaliar os fluxos de informação entre os métodos e classes, que não envolvam fluxos alternativos persistentes.

REFERÊNCIAS

BEIZER, B. Software **System Testing and Quality Assurance.** Van Nostrand Reinhold Company Inc, 1984.

BUZATO, Luiz Eduardo; RUBIRA, Cecília M. F. **Construção de sistemas orientados a objetos confiáveis.** Rio de Janeiro, 1998. 170p.

COAD, Peter; YOURDON, Edward. **Projeto baseado em objetos.** Editora Campus, Rio de Janeiro, Yourdon press, 1993. 195p.

DAVID, Marcio F. **Programação Orientada a Objetos: uma introdução**. Disponível em <http://www.guiadohardware.net/artigos/programacao-orientada-objetos/>. Acesso em: 14 de março de 2008.

DEITEL, H. M.; DEITEL, P. J. **Java : como programar**. 4ª ed. Porto Alegre: Bookman, 2005.

DELAMARO, Marcio E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software** . Editora Campus, 2007.

DOMIGUES, Andre L.S. **Avaliação de Critérios e Ferramentas de Teste para programas OO.** Dissertação de mestrado (Mestre em matermática computaciona) - Instituto de Ciencias Matemáticas e Computação de Sao Carlos - Sao Carlos, 2002.

HARROLD, M. J; ROTHERMEL, G., **Performing Data Flow Testing on Classes.** Proc. of the 2nd ACM SIGSOFT Symposium on Foundations of Soft. Eng., Vol. 19, N. 5, Dezember 1994, pp. 154-163.

LIMA, Gladys Machado Pereira Santos . TRAVASSOS, Guilherme Horta. **Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes.** Programa de Engenharia de Sistemas e Computação- COPPE / UFRJ; Rio de Janeiro, 2004 .

MARTIN, James; ODELL, James J. **Análise e Projeto Orientado a Objeto**; São Paulo; Makron Books 1995.

MARTINS, Jefferson Carlos; TSCHANNERL, Helbert Luiz. **Testes de Software Aplicado à Orientação a Objetos**. 2003. Disponível em <http://www.pr.gov.br/batebyte/edicoes/2003/bb136/testes.shtml>. Acesso em 24 de março de 2008.

MCGREGOR, J.D.; KORSONS, T.D. **Integrated Object- Oriented Testing and Development Processes, Communications of the ACM**. September 1994, vol. 37, nº 39, page(s): 59-77.

MOLINA, G. B. **A Eficácia de Critérios de Teste Estrutural em Aplicação de Banco de Dados Relacional** .Dissertação de Mestrado – UNIVEM – Marília – SP – 2005

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

NARDI, Paulo Augusto. **Inclusão do Critério Todos T-Usos na Ferramenta Jabuti.** Dissertação de mestrado. Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, 2006.

PRESSMAN, Roger S. **Engenharia de software**. São Paulo: Makron Books, 5º edição, 1995.

RAPPS, S.; WEYUKER, E.J. **Data Flow Analysis Techniques for Test Data Selection.** International Conference on Software Engineering, Tokio, Japan, Sepetember, 1982.

RAPPS, S.; WEYUKER, E.J. **Selecting Software Test Data Using Data Flow Information**. IEE Transactions on Software Engineering, SE-11, abril, 1985.

RIBEIRO, Ricardo Lopes. **Testes de Software: Uma Visão para Aplicações Orientadas a Objeto.** Disponível em <http://www.mundooo.com.br/php/modules.php?name=MOOArtigos&pa=showpage&pid=11> Acesso em 27/03/2008.

RICARTE, Ivan Luiz Marques. **Programação Orientada a Objetos: Uma Abordagem com Java**. Faculdade de Engenharia Elétrica e de Computação- Universidade Estadual de Campinas, 2001.

SILVA , Elvio Gilberto. **Análise Orientada a Objeto: Conceitos do Paradigma de Orientação a Objetos**. FMR – Faculdade Marechal Rondon Gestão de Sistemas de Informação. São Paulo, 2008.

SILVEIRA, I. F. **Linguagem Java.** 2003; Disponível em <http://www.infowester.com/lingjava.php>. Acesso em: 3 de novembro de 2008.

SOMMERVILLE, Ian**. Engenharia de software**. 6ª ed, São Paulo: Addison-Wesley, 2004. 592p.

SPOTO, E.S. **Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional**. Tese de Doutorado, UNICAMP, Campinas, Brasil, 2000

VICENZI, Auri M. R. **Orientação a objeto: Definição, Implementação e Análise de Recursos de Teste e Validação**. Dissertação de doutorado; ICMC-USP São Carlos, 2004.

ANEXO A - Script de criação do banco de dados implementado para a realização do teste.

```
// TABELA DE USUARIO DA BIBLIOTECA
CREATE TABLE USUARIO
     (USU_CPF    NUMBER(11),
     USU_NOME    VARCHAR2(40) NOT NULL,
     USU_END     VARCHAR(50) NOT NULL,
     USU_BAIRRO  VARCHAR(20),
     USU_CIDADE  VARCHAR(30) NOT NULL,
     USU_UF          VARCHAR(02),
     USU_CEP         VARCHAR(08),
     USU_TEL         VARCHAR(15),
     USU_EMAIL   VARCHAR(40),
     USU_SENHA   VARCHAR(08),
     CONSTRAINT  PK_USU_CPF PRIMARY KEY (USU_CPF)
     );

// TABELA DE EDITORA DA BIBLIOTECA
CREATE TABLE EDITORA
     (EDT_COD     NUMBER(06),
     EDT_NOME     VARCHAR2(40) NOT NULL,
     EDT_END      VARCHAR(50) NOT NULL,
     EDT_BAIRRO   VARCHAR(20),
     EDT_CIDADE   VARCHAR(30) NOT NULL,
     EDT_UF           VARCHAR(02),
     EDT_CEP          VARCHAR(08),
     EDT_TEL          VARCHAR(15),
     EDT_FAX          VARCHAR(15),
     EDT_EMAIL    VARCHAR(40),
     EDT_HOMEPAGE VARCHAR(40),
     CONSTRAINT  PK_EDT_COD PRIMARY KEY (EDT_COD)
     );
ALTER TABLE EDITORA
      MODIFY EDT_HOMEPAGE VARCHAR(40);

// TABELA DE AUTOR DA BIBLIOTECA
CREATE TABLE AUTOR
     (AUT_COD     NUMBER(06),
```

```
        AUT_NOME     VARCHAR2(40) NOT NULL,
        CONSTRAINT  PK_AUT_COD PRIMARY KEY (AUT_COD)
        );

    // TABELA DE LIVRO DA BIBLIOTECA
    CREATE TABLE LIVRO
        (LIV_COD       VARCHAR(15),
        LIV_TITULO     VARCHAR2(40) NOT NULL,
        LIV_IDIOMA     VARCHAR(15) NOT NULL,
        LIV_QTEXEMPLAR NUMBER(06),
        LIV_PRZENTREGA NUMBER(03),
        LIV_QTDISPON   NUMBER(04),
        LIV_EDT_COD    NUMBER(06),
        LIV_VOLUME     VARCHAR(04),
        LIV_ANOEDICAO  NUMBER(04),
        LIV_NUMEDICAO  NUMBER(04),
        LIV_TRADUTOR   VARCHAR(40),
        CONSTRAINT PK_LIV_COD     PRIMARY KEY (LIV_COD),
        CONSTRAINT FK_LIV_EDT_COD FOREIGN KEY (LIV_EDT_COD)
           REFERENCES EDITORA(EDT_COD)
        );
    // TABELA DE EXEMPLAR DO LIVRO DA BIBLIOTECA
    CREATE TABLE EXEMPLAR
        (EXP_LIV_COD   VARCHAR(15),
        EXP_COD        NUMBER(06),
        EXP_TOMBO      VARCHAR(15) NOT NULL,
        EXP_DTCOMPRA   DATE,
        EXP_VLCOMPRA   NUMBER(10,2),
        EXP_EMPRESTA   VARCHAR(01),
        CONSTRAINT        PK_EXP_LIV_COD_EXP_COD                PRIMARY
KEY(EXP_LIV_COD,EXP_COD),
        CONSTRAINT FK_EXP_LIV_COD FOREIGN KEY(EXP_LIV_COD)
           REFERENCES LIVRO(LIV_COD)
        );
    // TABELA LIVRO_AUTOR
    CREATE TABLE LIVRO_AUTOR
        (LVA_LIV_COD   VARCHAR(15),
        LVA_AUT_COD    NUMBER(06),
        CONSTRAINT        PK_LVA_LIV_COD_AUT_COD                PRIMARY
KEY(LVA_LIV_COD,LVA_AUT_COD),
        CONSTRAINT FK_LVA_LIV_COD FOREIGN KEY(LVA_LIV_COD)
```

```
   REFERENCES LIVRO(LIV_COD),
CONSTRAINT FK_LVA_AUT_COD FOREIGN KEY(LVA_AUT_COD)
   REFERENCES AUTOR(AUT_COD)
);
```

ANEXO B – Código fonte das principais classes utilizadas para este trabalho: classe DataBase e Usuário.

```
class DataBase
{
     //classe DataBase
     private static Connection connection;
     //para conectar apenas uma vez (estático)
     private static DataBase instance = null;

public DataBase() {
     super();
}
public static Connection getConection()
{
     return connection;
}
public static void setConection(Connection p_conection)
{
     connection = p_conection;
}
public static void loadDriver()
{
     //carregando drive database
     if (instance == null)
     {
           instance = new DataBase();
           try
                 {
                 System.out.println("Carregando Drive Database...");
                 String url = "jdbc:odbc:nivia";

   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
                 System.out.println("Ok Driver odbc Carregado");

                 Connection                conexao                =
DriverManager.getConnection(url,"scott","tiger");

                 System.out.println("Ok DriverManager");
                 //conexao.setAutoCommit(true);
```

```
                System.out.println("Ok setAutoCommit");
                setConection(conexao);
                System.out.println("Ok setConection");
          }
          catch (Exception e)
          {
                System.out.println("Error loadDriver--> "+e);
          }
     }//fim do if instance=null
}// fim loadDriver

/////abaixo insert/consulta/update/delete Para todas os bds
public boolean insertDB(String sql)
{
     boolean okInsert = false;
     Statement stm;
     try
     {
            //mostrando a sql no debug
            System.out.println(sql);
            Connection conexao=null;
               //criando o objeto Statement stm
            stm = getConection().createStatement();
            //executando a sql
            okInsert = stm.execute(sql);
            System.out.println(" Dados inseridos com sucesso!!!");
            conexao.commit();
     }
     catch (SQLException sqlEx)
     {
            System.err.println("Erro InsertDB--> "+sqlEx);
     }
     //retornando se ok sql
     return okInsert;
}

public ResultSet consultaDB(String sql)
{
     Statement stm;
     ResultSet rSet=null;
     try
```

```
    {
            //mostrando a sql no debug
            System.out.println(sql);
        //criando o objeto Statement stm
            stm = getConection().createStatement();
            //executando a sql
            rSet = null;
            rSet = stm.executeQuery(sql);
            System.out.println("Consultando o banco...");
            System.out.println(rSet);
    }
    catch (SQLException sqlEx)
    {
            System.err.println("Erro consultaDB--> "+sqlEx);
    }
    return rSet;
}
public boolean deleteDB(String sql)
{
    boolean okDelete = false;
    Statement stm;
    try
    {
            //mostrando a sql no debug
            System.out.println(sql);
        //criando o objeto Statement stm
            stm = getConection().createStatement();
            //executando a sql
            okDelete = stm.execute(sql);
            Connection conexao=null;
            System.out.println(" Excluído com sucesso");
            conexao.commit();
    }
    catch (SQLException sqlEx)
    {
            System.err.println("Erro DeleteDB--> "+sqlEx);
    }
    //retornando se ok sql
    return okDelete;
}
public boolean updateDB(String sql)
```

```
            {
                  boolean okUpdate = false;
                  Statement stm;
                  try
                  {
                            //mostrando a sql no debug
                            System.out.println(sql);
                            //criando o objeto Statement stm
                            stm = getConection().createStatement();
                            System.out.println("Passou    create    Statement
update");
                            //executando a sql
                            okUpdate = stm.execute(sql);
                            Connection conexao=null;
                            conexao.commit();
                            System.out.println("Dados alterados");
                            System.out.println("Passou execute update");
                  }
                  catch (SQLException sqlEx)
                  {
                        System.err.println("Erro updateDB--> "+sqlEx);
                  }

             //retornando se ok sql
             return okUpdate;
            }

        }// fim classe DataBase

        ////////////////////////////
        ///classe usuario

        class Usuario {
             private long USU_CPF;
             private String USU_NOME;
             private String USU_END;
             private String USU_BAIRRO;
             private String USU_CIDADE;
             private String USU_UF;
             private String USU_CEP;
             private String USU_TEL;
```

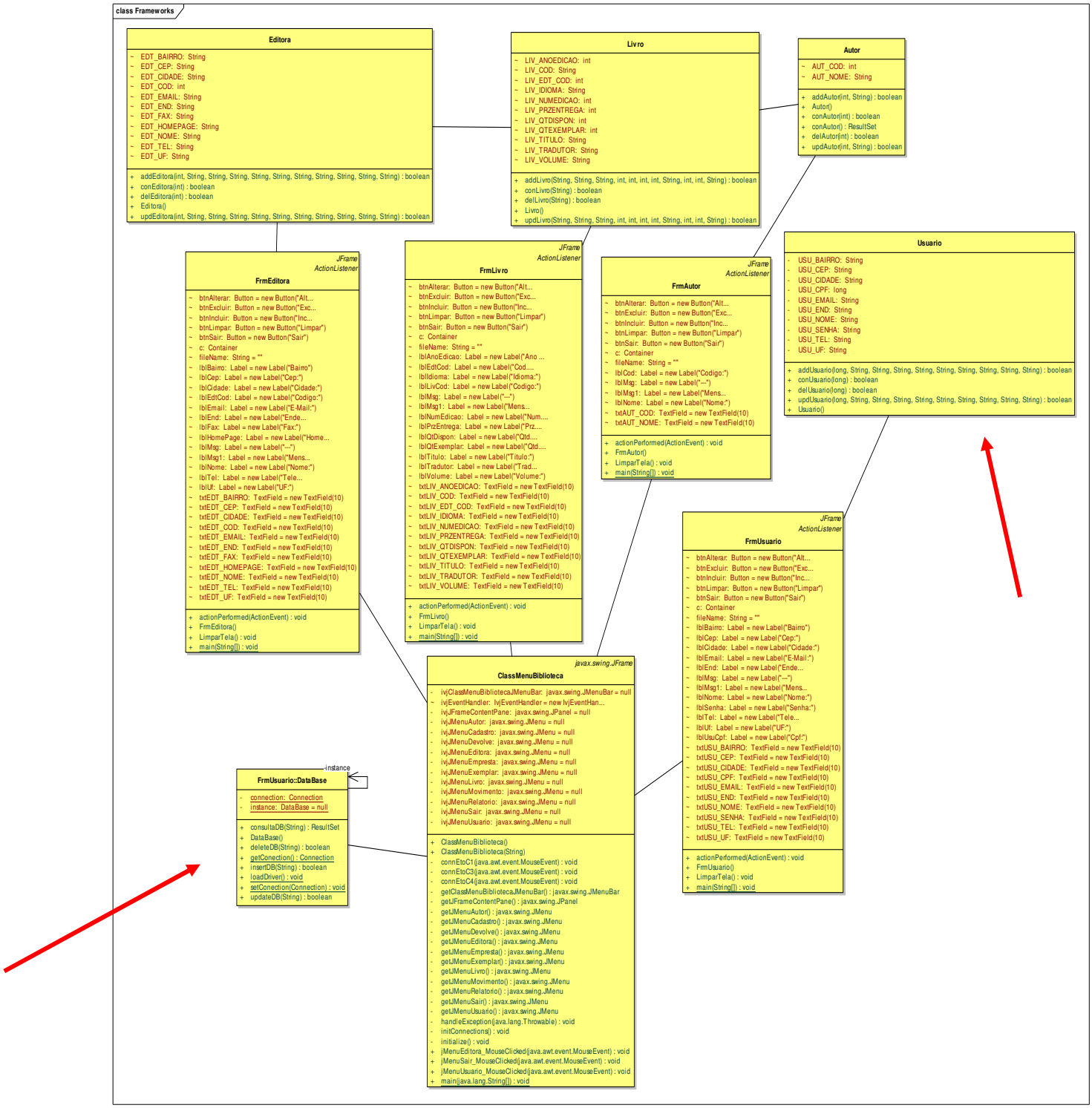
```
        private String USU_EMAIL;
        private String USU_SENHA;

    public Usuario() {
        super();
    }
        public boolean addUsuario(
                    long pUSU_CPF,
                    String pUSU_NOME,
                    String pUSU_END,
                    String pUSU_BAIRRO,
                    String pUSU_CIDADE,
                    String pUSU_UF,
                    String pUSU_CEP,
                    String pUSU_TEL,
                    String pUSU_EMAIL,
                    String pUSU_SENHA)
        {
              boolean sucesso = false;
              try
              {
                    String sql = null;
                    //montando a SQL
                    sql       =       "INSERT       INTO       USUARIO
(USU_CPF,USU_NOME,USU_END,";
                    sql              =              sql              +
"USU_BAIRRO,USU_CIDADE,USU_UF,USU_CEP,";
                    sql = sql + "USU_TEL,USU_EMAIL,USU_SENHA)";
                    sql         =         sql         +         "VALUES
("+pUSU_CPF+",'"+pUSU_NOME+"','"+pUSU_END+"','";
                    sql              =              sql              +
      pUSU_BAIRRO+"','"+pUSU_CIDADE+"','"+pUSU_UF+"','";
                    sql              =              sql              +
pUSU_CEP+"','"+pUSU_TEL+"','"+pUSU_EMAIL+"','"+pUSU_SENHA+"')";
                    DataBase db = new DataBase();
                    System.out.println("sql insert USU->"+ sql);
                    //método  loadDriver  e  insertDB  são  da  classe
DataBase
                    db.loadDriver();
                    sucesso = db.insertDB(sql);
```

```
                }
                catch (Exception e)
                {
                        System.out.println("Erro Usuario.addUsuario->"+ e);
                }
                return sucesso;
        }
        public boolean conUsuario(long pUSU_CPF)
        {
                boolean sucesso = false;
                ResultSet rSet = null;
                try
                {
                        String sql = null;
                        //montando a SQL
                        sql = "SELECT * FROM USUARIO WHERE USU_CPF = "
+pUSU_CPF;
                        DataBase db = new DataBase();
                        //métodos loadDriver e insertDB são da classe
DataBase
                        db.loadDriver();
                        rSet = db.consultaDB(sql);
                        sucesso = false;
                        while (rSet.next())
                        {
                                System.out.println("passou                aqui
USU==>"+rSet.getString("usu_nome"));
                                sucesso = true;
                        }
                        if (rSet != null) rSet.close();
                }
                catch (Exception e)
                {
                        System.out.println("Erro Usuario.conUsuario->"+ e);
                }
                return sucesso;
        }
        public boolean updUsuario(
                        long pUSU_CPF,
                        String pUSU_NOME,
                        String pUSU_END,
```

```
                        String pUSU_BAIRRO,
                        String pUSU_CIDADE,
                        String pUSU_UF,
                        String pUSU_CEP,
                        String pUSU_TEL,
                        String pUSU_EMAIL,
                        String pUSU_SENHA)
                  {
                  boolean sucesso = false;
                  try
                  {
                        String sql = null;
                        //montando a SQL
                        System.out.println("nome usuario"+pUSU_NOME);
                        sql    =    "UPDATE    USUARIO    SET    USU_NOME    =
'"+pUSU_NOME+"', ";
                        sql = sql +"USU_END = '"+pUSU_END+"',";
                        sql = sql +"USU_BAIRRO = '"+pUSU_BAIRRO+"',";
                        sql = sql +"USU_CIDADE = '"+pUSU_CIDADE+"',";
                        sql = sql +"USU_UF = '"+pUSU_UF+"',";
                        sql = sql +"USU_CEP = '"+pUSU_CEP+"',";
                        sql = sql +"USU_TEL = '"+pUSU_TEL+"',";
                        sql = sql +"USU_EMAIL = '"+pUSU_EMAIL+"',";
                        sql = sql +"USU_SENHA = '"+pUSU_SENHA+"' ";
                        sql = sql +" WHERE USU_CPF = "+pUSU_CPF;
                        DataBase db = new DataBase();
                        //métodos  loadDriver  e  insertDB  são  da  classe
DataBase
                        db.loadDriver();
                        sucesso = db.updateDB(sql);
                  }
                  catch (Exception e)
                  {
                        System.out.println("Erro
Funcionario.updFuncionario->"+ e);
                  }
                  return sucesso;
            }

            public boolean delUsuario(long pUSU_CPF)
            {
```

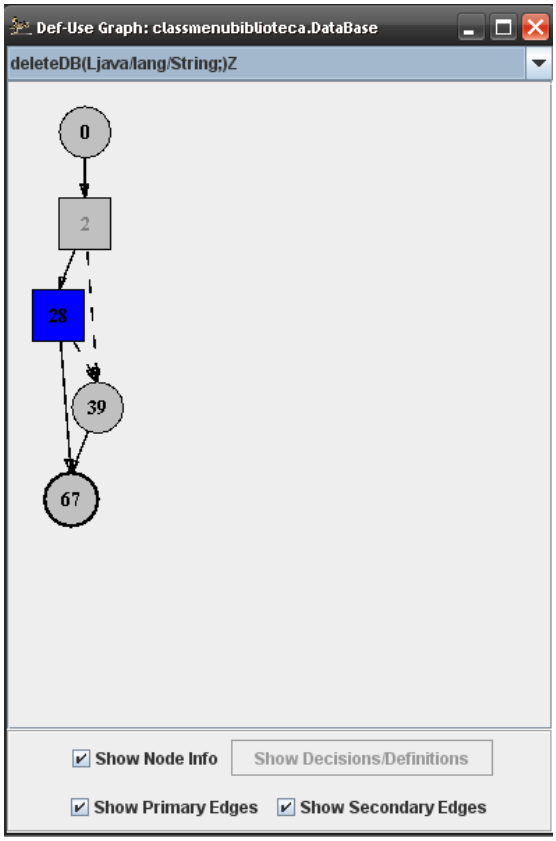
```
                boolean sucesso = false;
                try
                {
                      String sql = null;
                      //montando a SQL
                      sql = "DELETE FROM USUARIO WHERE USU_CPF =
"+pUSU_CPF ;
                      DataBase db = new DataBase();
                      //método loadDriver e insertDB são da classe
DataBase
                      db.loadDriver();
                      sucesso = db.deleteDB(sql);
                }
                catch (Exception e)
                {
                      System.out.println("Erro
Funcionario.delFuncionario->"+ e);
                }
                return sucesso;
            }

       }//fim classe usuario
```

ANEXO C - Diagrama de classe do Sistema Biblioteca utilizado neste trabalho. As principais classes utilizadas estão destacadas nesta figura abaixo

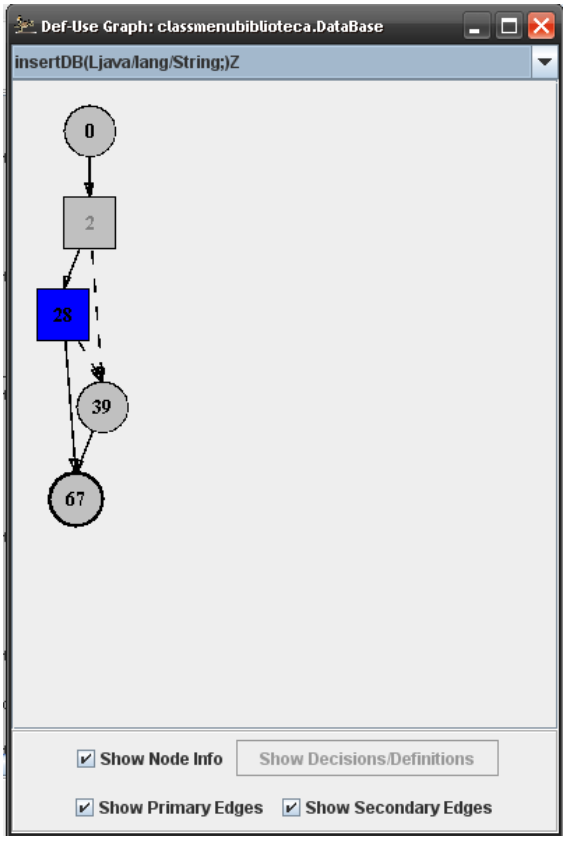


ANEXO D - Grafos dos metodos da classe Database que foram utilizados para realizar a estratégia de teste.

Grafo do método deleteDB



Grafo do método insertDB



Grafo do método updateDB