DENERVAL MENDEZ BATISTA

DBValTool: UMA FERRAMENTA PARA APOIAR O TESTE E A VALIDAÇÃO DE PROJETO DE BANCO DE DADOS RELACIONAL

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática pelo Curso de Pós-Graduação em Informática, do Setor de Ciências Exatas da Universidade Federal do Paraná, em convênio com o Departamento de Informática da Universidade Estadual de Maringá.

Orientador: Dr. Márcio Eduardo Delamaro

Orientador: Dr. Edmundo Sérgio Spoto

CURITIBA

2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

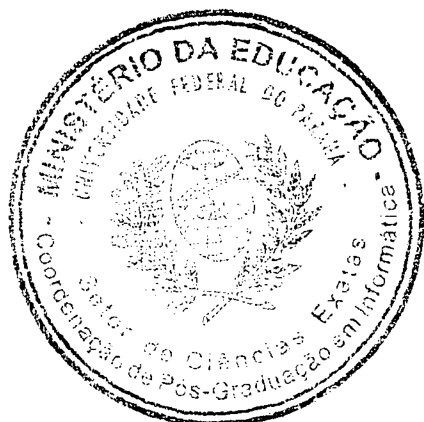
Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno *Denerval Mendez Batista* , avaliamos o trabalho intitulado, "*DBVALTOOL: Uma Ferramenta para Apoiar o Teste e a Validação de Projeto Lógico de Banco de Dados Relacional* ", cuja defesa foi realizada no dia 29 de agosto de 2003, às dez horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato. (Convênio número 279-00/UFPR de Pós-Graduação entre a UFPR e a UEM - ref. UEM número 1331/2000-UEM).

Curitiba, 29 de agosto de 2003.

Prof. Dr. Edmundo Sérgio Spoto
UNIVEM – Orientador

Prof. Dr. Márcio Eduardo Delamaro
UNIVEM – Orientador

Prof. Dr. Mauro Biajiz
DC/UFSCAR – Membro Externo



Prof^a. Dra. Silvia Regina Vergilio
DINF/UFPR – Membro Interno

AGRADECIMENTOS

A Deus, que sempre me acompanhou, deu-me saúde e permitiu que eu pudesse realizar este trabalho.

À Cláudia – meu amor, minha vida e minha esposa –, pelo amor, carinho e apoio recebidos durante a nossa vida e, principalmente, durante o período em que realizei este trabalho.

Aos meus filhos Felipe e Ana Luísa que, apesar de pequenos e não entenderem o que significa este trabalho, ajudaram simplesmente por existirem.

Aos meus pais Paulo e Clarice, pelo amor e educação recebidos durante toda a minha vida.

À minha irmã Sueleni, pelo apoio e companheirismo demonstrados durante a realização desse curso de pós-graduação.

Ao meu co-orientador Edmundo Sérgio Spoto, principal companheiro na realização deste trabalho, por sempre acreditar e confiar no meu trabalho, e pelo respeito, profissionalismo e companheirismo demonstrados durante o período em que trabalhamos juntos; qualidades essas que, reunidas, conheci em poucas pessoas.

Ao meu orientador Márcio Eduardo Delamaro, pelas contribuições a este trabalho e pelo respeito e profissionalismo demonstrados durante o período em que trabalhamos juntos.

Aos amigos da CVU-UEM, pelo apoio recebido e pela compreensão nos momentos de ausência; ao amigo Juliano, do DLE-UEM, pela ajuda com a Língua Portuguesa e aos demais amigos que, de maneira direta ou indireta, contribuíram para a realização deste trabalho e torceram pelo meu sucesso.

A atividade de teste nunca termina;
ela apenas é transferida do projetista
para o seu cliente. (...)

Frase atribuída aos projetistas de *software* experientes, por Roger S. Pressman.

Estamos trabalhando para que a atividade
de teste termine e que o produto entregue
ao cliente não possua defeitos.

Denerval Mendez Batista

SUMÁRIO

LISTA DE FIGURAS ... *vi*

LISTA DE QUADROS ... *vii*

LISTA DE ABREVIATURAS E SIGLAS ... *viii*

RESUMO ... *ix*

ABSTRACT ... *x*

1 INTRODUÇÃO ... 1

1.1 SITUAÇÃO DO PROBLEMA ... 1

1.2 OBJETIVOS ... 3

1.3 TRABALHOS RELACIONADOS ... 3

1.4 ORGANIZAÇÃO DO TRABALHO ... 5

2 BANCO DE DADOS RELACIONAL: CONCEITOS E TERMINOLOGIA ... 6

2.1 RESTRIÇÕES DE INTEGRIDADE ... 8

2.1.1 Restrições de Integridade Básica ... 8

2.1.2 Restrições de Integridade Semântica ... 10

2.2 SQL E ESPECIFICAÇÃO DE RESTRIÇÕES DE INTEGRIDADE EM SQL ... 12

2.3 *TRIGGERS* E BANCOS DE DADOS ATIVOS ... 13

2.4 PROJETO DE BANCO DE DADOS ... 15

2.4.1 Definição de Requisitos ... 15

2.4.2 Projeto Conceitual ... 16

2.4.3 Projeto Lógico ... 16

2.4.4 Projeto Físico ... 16

2.4.5 Problemas em Projeto de Banco de Dados ... 17

2.5 DIAGRAMA ENTIDADE-RELACIONAMENTO (DER) ... 17

2.5.1 Entidade ... 18

2.5.2 Relacionamento ... 18

2.5.3 Atributo ... 19

2.6 MAPEAMENTO OBJETO-RELACIONAL ... 19

2.7 CONSIDERAÇÕES FINAIS ... 20

3 TESTE E VALIDAÇÃO: CONCEITOS E TERMINOLOGIA ... 21

3.1 A ATIVIDADE DE TESTE E O TESTE DE BASES DE DADOS ... 21

3.1.1 Conceitos em Teste de Bases de Dados ... 22

3.1.2 O Estado do Banco de Dados Durante o Teste 22
3.2 AS ATIVIDADES DE VERIFICAÇÃO E VALIDAÇÃO (V&V) 23
3.2.1 Verificação de Requisitos 24
3.2.2 Verificação de Projeto 25
3.2.3 Verificação de Código 26
3.2.4 Validação 26
3.3 CONSIDERAÇÕES FINAIS 27
4 O MODELO PROPOSTO E A FERRAMENTA DBValTool 28
4.1 PROPOSTA DE MODELO DE PROCESSO DE VALIDAÇÃO DE PROJETO ... 28
4.2 A FERRAMENTA DBValTool 31
4.2.1 Aspectos Funcionais 32
4.2.2 Arquitetura 33
4.2.3 Implementação 36
4.2.4 *Warnings* de Projeto de Banco de Dados Propostos e Implementados 40
4.3 CONSIDERAÇÕES FINAIS 43
5 VALIDAÇÃO DO MODELO PROPOSTO E DA FERRAMENTA DBValTool 45
5.1 METODOLOGIA 45
5.2 PROCEDIMENTOS 46
5.3 RESULTADOS 48
5.4 CONSIDERAÇÕES FINAIS 50
6 CONCLUSÕES E TRABALHOS FUTUROS 51
6.1 CONCLUSÕES 51
6.2 CONTRIBUIÇÕES DESTE TRABALHO 52
6.3 TRABALHOS FUTUROS 53
REFERÊNCIAS 55
DOCUMENTOS CONSULTADOS 57
ANEXOS 58
ANEXO 1 - UML 58
ANEXO 2 - TESTE DE PROGRAMAS 64

LISTA DE FIGURAS

FIGURA 1 – EXEMPLO DE NOTAÇÃO DE ENTIDADES EM UM DER 18

FIGURA 2 – EXEMPLO DE NOTAÇÃO DE RELACIONAMENTO EM UM DER 18

FIGURA 3 – MODELO DE PROCESSO DE VALIDAÇÃO DE PROJETO 29

FIGURA 4 – DIAGRAMA DE *USE CASES* DA DBValTool 33

FIGURA 5 – ARQUITETURA DA DBValTool 34

FIGURA 6 – DIAGRAMA DE CLASSES DA DBValTool 37

FIGURA 7 – DIAGRAMA ENTIDADE-RELACIONAMENTO (DER) DA DBValTool 38

FIGURA 8 – UMA VISÃO DO GERENCIAMENTO DE SESSÕES DE TESTE NA DBValTool 39

FIGURA 9 – UMA VISÃO DA EXECUÇÃO DE UM CASO DE TESTE NA DBValTool 40

FIGURA 10 – EXEMPLO DE *WARNING* DO TIPO W06 43

FIGURA 11 – EXEMPLO DE *WARNING* DO TIPO W07 43

FIGURA 12 – NOTAÇÃO UML PARA UM DIAGRAMA DE *USE CASES* 61

FIGURA 13 – NOTAÇÃO UML PARA AS CLASSES 62

FIGURA 14 – EXEMPLOS DOS TIPOS DE RELACIONAMENTOS NOS DIAGRAMAS DE CLASSES, UTILIZANDO NOTAÇÃO UML .. 63

LISTA DE QUADROS

QUADRO 1 – RESUMO QUANTITATIVO DAS DEFINIÇÕES E DECLARAÇÕES FORMAIS ENCONTRADAS NOS PROJETOS 46

QUADRO 2 – RESUMO QUANTITATIVO DOS RESULTADOS OBTIDOS 48

QUADRO 3 – TIPOS DE DEFEITOS DETECTADOS COM O APOIO DA DBValTool 49

QUADRO 4 – NOTAÇÕES DE CARDINALIDADE 62

LISTA DE ABREVIATURAS E SIGLAS

ABDR	–	Aplicação de Banco de Dados Relacional
DDL	–	*Data Definition Language*
DER	–	Diagrama Entidade-Relacionamento
DML	–	*Data Manipulation Language*
EC	–	Estudo de Caso
OCL	–	*Object Constraint Language*
ODBC	–	*Open Database Connectivity*
OMT	–	*Object Modeling Technique*
OOSE	–	*Object-Oriented Software Engineering*
RI	–	Restrição de Integridade
RUP	–	*Rational Unified Process*
SGBD	–	Sistema Gerenciador de Banco de Dados
SGBD-OO	–	Sistema Gerenciador de Banco de Dados Orientado a Objeto
SGBDR	–	Sistema Gerenciador de Banco de Dados Relacional
SQL	–	*Structured Query Language*
UML	–	*Unified Modeling Language*
V&V	–	Verificação e Validação

RESUMO

A capacidade de armazenar, disponibilizar e proteger uma grande quantidade de dados confere aos bancos de dados papel de grande importância na maioria das modernas organizações, sejam elas grandes ou pequenas. É essencial que os sistemas de banco de dados funcionem corretamente e apresentem desempenho aceitável. Diante da importância da integridade e da consistência das informações a serem armazenadas nos sistemas de banco de dados, técnicas específicas para testar as bases de dados, principalmente antes de serem povoadas, respondem a uma necessidade muitas vezes vital. Nesse sentido, este trabalho apresenta um modelo de processo de validação de projeto, que tem como objetivo principal aumentar a qualidade e a confiabilidade dos projetos de banco de dados. A Ferramenta DBValTool, que pode ser utilizada em várias etapas do modelo proposto, foi desenvolvida para fornecer o apoio necessário ao teste de bases de dados, buscando proporcionar um aumento da qualidade e uma redução de tempo e de custo durante a atividade de teste. O uso do modelo proposto e da Ferramenta DBValTool, em estudos de casos realizados, apresentou, como se esperava inicialmente, contribuições ao processo de teste e de validação do projeto.

ABSTRACT

The capacity to store, to make available and to protect a large amount of data gives the database a role of great importance in most modern organizations. It is essential that the database system properly works and shows an acceptable performance. Considering the importance of both integrity and consistency of the information to be stored into the database systems, specific techniques to test the databases, mainly before being filled, are essentials. This study shows a process model for design validation that aims at enhancing both the quality and reliance of the database projects. The DBValTool, which can be used in several stages of the suggested model, was developed to give the necessary support to the databases test, trying to improve the quality and reduce both the time and cost during the test activity. The use of the suggested model and the DBValTool in case studies performed, showed, contributions to the validation and test process of the database design.

1 INTRODUÇÃO

Neste capítulo, são apresentados o contexto e os objetivos deste trabalho, bem como alguns trabalhos relacionados. A organização da dissertação é apresentada na última seção deste capítulo.

1.1 SITUAÇÃO DO PROBLEMA

Apesar dos significativos avanços obtidos, nos últimos anos, nas diversas áreas da computação, defeitos, causados na maior parte por falhas humanas, estão presentes em muitos produtos de *software* produzidos e liberados para o uso. Diante dos esforços no sentido de revelar esses defeitos, antes que o produto seja utilizado pelo cliente, a atividade de teste tem-se mostrado uma grande aliada. Por meio do teste, tem-se conseguido um aumento da qualidade e da confiabilidade de um produto (PRESSMAN, 1998).

O teste é uma atividade importante na avaliação de programas, de programas de banco de dados e, também, dos esquemas das bases de dados por meio de *queries* em comandos de SQL (*Structured Query Language*).

Tradicionalmente, o teste de programas tem sido o foco principal do esforço de pesquisa e de desenvolvimento. O teste de programa consiste em exercitar um programa por meio de dados de entrada e comparar os valores de saída produzidos com o objetivo de encontrar defeitos (MYERS, 1979), tanto na sua estrutura quanto nas suas funcionalidades.

Diante do importante papel que os bancos de dados desempenham nas modernas organizações – armazenando, disponibilizando e protegendo uma grande quantidade de dados – e da importância da integridade e da consistência das informações a serem armazenadas nos sistemas de banco de dados, técnicas específicas para testar as bases de dados, principalmente antes de serem povoadas (CHAYS et al., 2000), respondem a uma necessidade muitas vezes vital.

No teste de base de dados, a estrutura lógica dos dados passa a ser o objeto do teste e o processo consiste em executar operações sobre uma base de dados, visando à ocorrência de falhas (ARANHA et al., 2000). Nesse teste é essencial exercitar as necessidades especificadas da base de dados, de maneira que o projeto de banco de dados implementado atenda essas necessidades com segurança, para futuramente contemplar todas as funcionalidades das aplicações que se utilizam dessa base de dados. Sugere-se que um projeto de aplicação de banco de dados seja iniciado com uma avaliação da base de dados, a fim de garantir que essa base esteja validada, para, só então, partir para o desenvolvimento da aplicação (SPOTO, 2000).

Fazendo um paralelo entre teste de programas e teste de bases de dados, a descrição daquele somente se aplica a este se considerarmos o teste de aplicações que fazem uso de bases de dados ou o teste de eventuais elementos associados aos dados tais como *triggers*, regras ativas ou procedimentos.

As atividades de teste consomem, em média, 40% do tempo e do custo de desenvolvimento de um produto de *software* (PRESSMAN, 1998). Esse gasto é, em parte, resultante da escassez de ferramentas de auxílio ao teste que permitam uma forma planejada e segura para sua execução e para a avaliação de seus resultados. Assim como nas outras áreas, a área de teste vem se utilizando cada vez mais de ferramentas automatizadas para apoiar as suas atividades. As ferramentas de teste, em sua maioria, reduzem o tempo de teste e aumentam a sua eficácia (PRESSMAN, 1998).

É importante esclarecer que, neste trabalho, os estudos realizados se limitam aos bancos de dados relacionais, uma vez que, independentemente dos motivos, ainda são os mais utilizados nas organizações. Informa-se, ainda, que a opção por manter alguns termos em inglês como, por exemplo, *trigger*, *function* e *procedure*, se deve ao fato de que uma grande parte dos desenvolvedores de sistemas de banco de dados assim utilizam esses termos.

1.2 OBJETIVOS

Diante das informações apresentadas anteriormente, verifica-se a necessidade do desenvolvimento e do aprimoramento de métodos e de técnicas, com o desejável apoio de ferramentas automatizadas, a fim de assegurar que o projeto de uma base de dados seja implementado da forma como a sua especificação determina. Assim, estudos no sentido de aumentar a qualidade e a confiabilidade de bases de dados respondem a essa necessidade.

Um modo sistemático de avaliar o quanto estão corretas e completas as definições criadas no banco de dados, visando detectar possíveis erros na criação dos esquemas relacionais, antes de passar para etapas seguintes do processo de desenvolvimento, pode contribuir para um aumento da qualidade e da confiabilidade de bases de dados. O modelo de processo de validação de projeto, apresentado neste trabalho, representa esforços nesse sentido.

Uma ferramenta para apoiar o teste de bases de dados pode contribuir para uma redução do tempo e do custo dessa atividade, aumentando a produtividade. A Ferramenta DBValTool (*DataBase Testing and Validation Tool*), apresentada neste trabalho, representa esforços nesse sentido. Ela foi desenvolvida para apoiar, também, o processo de validação de projeto, podendo ser utilizada em várias etapas do modelo proposto.

1.3 TRABALHOS RELACIONADOS

Atualmente, existem poucos trabalhos divulgados que tratam especificamente do teste de bases de dados. Acreditamos que essa deficiência é ocasionada pela falta de ferramentas e de técnicas apropriadas para esse tipo de teste. Alguns trabalhos que tratam de testes de bases de dados e testes em sistemas de banco de dados são apresentados a seguir.

MANNILA e RÄIHÄ (1989) propõem dois algoritmos para geração automática de uma base de dados relacional para teste de *queries*, mas não teste da

base de dados. Esse trabalho descreve uma técnica que possibilita criar uma amostra simplificada da base de dados real, de tal modo que represente todo o domínio válido. Em muitos casos, bases de dados menores podem facilitar a aplicação dos testes.

JAHNE, URBAN e DIETRICH (1996) apresentam um protótipo de ambiente para depuração de regras ativas que provê dois modos de visualização de regras ativas: modo de visualização para cada uma das regras e modo detalhado de visualização dos eventos, condições e ações existentes em cada regra.

CHAYS et al. (2000) apresentam uma proposta de como testar sistemas de banco de dados e programas de Aplicação de Banco de Dados Relacional (ABDR), na qual são analisados aspectos para a correção de um sistema de banco de dados. Dentre outras informações, esse trabalho apresenta alguns tipos de restrições que podem ser avaliadas (domínio, unicidade, valores de atributos, integridade referencial e integridade semântica). Outra observação importante apresentada está relacionada ao estado da base de dados antes e depois da execução de cada caso de teste, que não pode ser ignorado, em decorrência das suas implicações e conseqüências.

SPOTO (2000) apresenta técnicas apropriadas para o teste de ABDR que utilizam a linguagem de consulta SQL embutida em linguagens procedimentais. Nesse trabalho é proposta uma abordagem para o teste estrutural de programas ABDR e, devido à sua natureza, dois modelos de fluxo de dados são propostos: *intra-modular* e *inter-modular*. O modelo de fluxo de dados *intra-modular* aplica-se ao teste de unidade e ao teste de integração das unidades de um programa. O *inter-modular* aplica-se à integração dos programas que compõem uma aplicação. Critérios baseados nos modelos *intra-modular* e *inter-modular* são apresentados e discutidos com exemplos de sua aplicação. São também apresentados e discutidos resultados de experimentos. Esse trabalho aborda assuntos que devem ser tratados em uma fase posterior aos que serão apresentados neste trabalho.

ARANHA et al. (2000) apresentam técnicas para testar os esquemas das bases de dados. Tais técnicas visam exercitar, por meio de *queries*, os atributos e as restrições de integridade da base de dados. Nesse trabalho, várias famílias de critérios foram definidas e resultados de experimentos realizados com esses critérios são

apresentados. O trabalho de ARANHA et al. (2000) aborda assuntos tratados também no presente trabalho. Os dois trabalhos apresentam um objetivo em comum: testar os esquemas das bases de dados. Porém, este trabalho apresenta uma sistematização para o teste de base de dados e, também, a importância e o uso desse tipo de teste no processo de validação de projeto de banco de dados.

1.4 ORGANIZAÇÃO DO TRABALHO

Neste capítulo são apresentados, além do contexto e dos objetivos deste trabalho, alguns trabalhos relacionados.

O Capítulo 2 expõe conceitos e terminologia da área de Banco de Dados Relacional relacionados com projeto de banco de dados e com integridade e consistência dos dados.

No Capítulo 3, são apresentados conceitos e terminologia da área de Engenharia de Software relacionados às atividades de teste e de verificação e validação (V&V).

No Capítulo 4, são apresentadas uma proposta de modelo de processo de validação de projeto de banco de dados e a Ferramenta DBValTool, desenvolvida para apoiar, também, o modelo proposto.

O Capítulo 5 exibe a metodologia, os procedimentos e os resultados do processo de validação do modelo proposto e da Ferramenta DBValTool.

No Capítulo, apresentam-se as conclusões deste trabalho, bem como as suas contribuições e as sugestões de trabalhos futuros.

Dois anexos completam este trabalho. O Anexo 1 apresenta uma breve exposição das principais características e notações da UML (*Unified Modeling Language*), utilizada durante o processo de desenvolvimento da Ferramenta DBValTool. O Anexo 2 apresenta um material complementar, a fim de auxiliar a compreensão e os estudos de teste de programas.

2 BANCO DE DADOS RELACIONAL: CONCEITOS E TERMINOLOGIA

Bancos de dados relacionais são baseados no modelo de dados relacional, que representa uma base de dados como uma coleção de relações (ELMASRI; NAVATHE, 2000). Informalmente, cada relação pode ser considerada como uma tabela. Porém, cabe esclarecer que o conceito de relação é ligeiramente diferente do conceito de tabela. Relação equivale à noção de conjunto, ou seja, um agrupamento de elementos sem repetição. Tabela equivale à noção de coleção, ou seja, um agrupamento de elementos no qual é permitida a repetição. Quando uma relação é tratada como uma tabela de valores, cada linha na tabela representa uma coleção de valores de dados relacionados. Esses valores podem ser interpretados como fatos que representam uma entidade do mundo real ou um relacionamento.

Na terminologia do modelo relacional formal, uma linha é denominada uma *tupla*, um título de uma coluna na tabela é um *atributo*. Os tipos de dados descrevem os tipos de valores de cada coluna e esses tipos de valores são chamados de *domínios* dos atributos (ELMASRI; NAVATHE, 2000).

Uma tupla é um conjunto de pares (atributo-valor) que define uma linha da tabela. Se imaginarmos uma tabela como sendo um conjunto, uma tupla seria um elemento deste conjunto.

Um atributo é um nome utilizado para representar um item de dado. Em uma tabela, representa uma coluna. Um atributo pode apresentar um valor condizente com o domínio associado a ele ou *null*, que indica ausência de valor ou valor desconhecido. Um atributo pode conter apenas um valor atômico, ou seja, um valor indivisível.

Um domínio *D* é um conjunto de valores atômicos. Um domínio é uma descrição lógica e física dos valores permitidos do atributo. Ele pode ser especificado por um tipo de dado cujos valores condizem com sua representação. Um domínio é estabelecido por um nome, um tipo de dado e um formato. Domínios podem ser simples, ou seja, possuem um único valor, como um inteiro, ou compostos, quando possuem vários valores, como uma data, que apresenta dia, mês e ano. Os valores dos itens de dados que apresentam domínios compostos são abstraídos e tratados como um

único valor. Uma data poderia ser manipulada como um *string*, por exemplo.

Um esquema de relação é usado para descrever uma relação. Um esquema de relação $R(A_1, A_2, ..., A_n)$ é um nome de relação R (nome de tabela) e um conjunto de atributos (nomes de colunas), cada um com um nome A_i e um domínio $dom(A_i)$.

Uma relação, ou estado da relação, de um esquema de relação R é um conjunto de tuplas, no qual cada uma é um elemento do produto cartesiano $dom(A_i)$ x ... x $dom(A_n)$.

Um esquema de relação descreve a estrutura dos dados, enquanto uma relação descreve o estado dos dados no momento atual. Um esquema de relação é definido no momento em que a base de dados é projetada e é pouco modificado, enquanto que o estado da relação é constantemente modificado para refletir as mudanças nas entidades do mundo real (ELMASRI; NAVATHE, 2000).

Cada esquema de relação ou tabela possui seus atributos e as respectivas restrições, de modo a estabelecer as regras de integridade entre as relações.

Um esquema da base de dados relacional $\boldsymbol{S}$ é um conjunto de esquemas de relação $S=\{R_1, R_2, .., R_m\}$ acompanhado de suas restrições de integridade.

O conceito de chave, segundo DATE (2000), é fundamental no modelo relacional para garantir a identificação de tuplas e estabelecer os relacionamentos entre as relações. O modelo relacional define dois tipos de chaves: chave primária e chave estrangeira.

A chave primária ou *pk* (*primary key*) de uma relação R é o atributo (ou grupo de atributos) A, em que todos os atributos de R são funcionalmente dependentes de A e não existe um subconjunto próprio de A que determine funcionalmente os atributos em R. Uma *pk* é escolhida dentre várias chaves candidatas, que podem estar presentes na relação. As chaves candidatas não selecionadas são ditas chaves alternativas.

A chave estrangeira ou *fk* (*foreign key*) é o atributo (ou grupo de atributos) de uma relação R_1 que estabelece um relacionamento de equivalência, por valor, com uma *pk* de uma relação R_2. O domínio da *fk* de R_1 deve ser compatível com o domínio da *pk* de R_2. Nada impede que R_1 e R_2 sejam a mesma relação.

2.1 RESTRIÇÕES DE INTEGRIDADE

Integridade é um conceito fundamental em banco de dados, uma vez que diz respeito à correção, consistência e segurança de dados armazenados. As restrições de integridade restringem os possíveis valores dos estados do banco de dados para refletir o mundo real com mais precisão. Há muitos tipos de restrições (ELMASRI; NAVATHE, 2000):

- Restrições de domínio: determinam possíveis valores de um atributo;
- Restrições de unicidade: determinam que duas tuplas distintas não podem ter os mesmos valores para um determinado conjunto de atributos;
- Restrições de não-nulo: determinam se um atributo pode ou não possuir valores nulos;
- Restrições de integridade referencial: determinam que valor de um atributo em uma relação R_1 deve também aparecer como valor do mesmo atributo na relação R_2;
- Restrições de integridade semântica: são restrições em valores do estado do banco de dados, expressos em alguma linguagem de especificação de restrições.

Diferentemente dos métodos tradicionais de modelagem de sistemas, os métodos mais modernos, principalmente os que dão enfoque à orientação a objetos, têm se preocupado em dar suporte às restrições. Isso pode ser visto na proposta de extensão à UML chamada OCL (*Object Constraint Language*).

2.1.1 Restrições de Integridade Básica

Os aspectos de integridade do modelo relacional estão associados aos conceitos de chave primária e chave estrangeira. Garantir a integridade de um esquema relacional significa garantir o acesso individualizado a todas as tuplas de uma relação, assim como garantir relacionamentos válidos e condizentes com a realidade. O modelo relacional é regido por duas regras de integridade básica (ELMASRI; NAVATHE, 2000): a regra de integridade de entidade e a regra de integridade referencial.

A regra de integridade de entidade diz respeito à chave primária de uma relação. Ela diz que nenhum atributo que faz parte da chave primária pode ter *null* em alguma tupla. A manutenção dessa regra garante que toda tupla possa ser identificada unicamente.

A regra de integridade referencial diz respeito às chaves estrangeiras de uma relação. Ela diz que o valor de um atributo que faz parte de uma chave estrangeira pode assumir *null* desde que o mesmo não faça parte da chave primária. Ainda, esse mesmo atributo pode assumir um valor qualquer, condizente com o seu domínio, desde que esse mesmo valor exista na chave primária de uma tupla da relação referida por ele. Com isto, nunca existirão relacionamentos incorretos entre dados.

Para que essas regras sejam sempre respeitadas, o Sistema Gerenciador de Banco de Dados (SGBD) deve implementar rotinas de verificação automáticas. Isso implica em testes e ações a serem realizados pelo SGBD toda vez que uma operação de atualização for submetida ao mesmo.

No caso da regra de integridade de entidade, o SGBD verifica e mantém esse tipo de integridade toda vez que for incluída uma nova tupla em uma relação ou for alterado o valor de um atributo de uma tupla que faz parte da chave primária de uma relação.

No caso da regra de integridade referencial, quando o SGBD percebe que uma violação nesse tipo de integridade irá ocorrer, uma ação previamente definida é tomada, dentre as três alternativas de ações possíveis:

- Impedimento (*restrict*): a operação não é efetivada;
- Cascata (*cascade*): para o caso de exclusões de tuplas ou alterações de chave primária na relação referida realiza-se a mesma operação em todas as tuplas que se referem a ela;
- Anulação (*set null*): para o caso de exclusões de tuplas ou alterações de chave primária na relação referida alteram-se para *null* o(s) atributo(s) que compõe(m) a chave estrangeira que estabelece o relacionamento com esta relação.

2.1.2 Restrições de Integridade Semântica

A integridade semântica é a garantia de que o estado dos dados do banco de dados está sempre coerente com a realidade para a qual o mesmo foi projetado e criado (ELMASRI; NAVATHE, 2000). A inexistência ou a falta de gerenciamento de integridade semântica pode causar, por exemplo, violação de domínio, ausência de valor em atributos e relacionamentos incorretos.

Um subsistema de integridade semântica de um SGBD controla restrições especificadas sobre um esquema. Essas restrições (ou regras) são chamadas restrições de integridade (RIs). Em toda RI deve ser possível especificar: para quais dados deve-se verificar a regra (ou alcance da regra), quando a regra deve ser verificada (antes, imediatamente após ou um tempo após a operação de atualização) e que ação deve ser tomada para garanti-la. Segundo ELMASRI e NAVATHE (2000), o gerenciamento dessas restrições envolve três funções básicas:

- Especificação de RIs: deve ser possível especificar testes e ações para garantia de integridade. Para tanto, o SGBD deve suportar uma linguagem de especificação de restrições (DCL), cujos comandos são compilados e mantidos no catálogo do sistema para posterior consulta e modificação;
- Monitoramento de RIs: sempre que ocorrer uma operação de atualização sobre um dado, todas as RIs que dizem respeito a esse dado devem ser consultadas, no catálogo do sistema, para verificar se alguma delas não foi violada ou se algum procedimento deve ser realizado;
- Ações para garantia de RIs: quando alguma operação de atualização gera alguma inconsistência, devem ser tomadas ações de impedimento da atualização (*rollback* da atualização) ou de execução de outros procedimentos para atualizar dados relacionados.

A existência de um subsistema de integridade semântica é extremamente vantajosa para as aplicações que utilizam um SGBD, pois existe total independência desse controle, ou seja, o gerenciamento de integridade fica a cargo do SGBD. As aplicações ficam liberadas dessa pesada tarefa, sendo que estas apenas submetem

operações ao SGBD e recebem indicação de que a operação foi concluída com sucesso ou sinalizações de violações e atitudes tomadas para garanti-las.

No que diz respeito aos bancos de dados relacionais, é possível classificar o universo possível de RIs semânticas em quatro categorias. Essa classificação é interessante, uma vez que todo SGBD deveria suportar a especificação e o controle de todas essas categorias (ELMASRI; NAVATHE, 2000).

- RIs quanto ao alcance: essa categoria classifica RIs de acordo com a quantidade (ou volume) de dados que deve ser consultada para que a regra seja verificada. À medida que aumenta a quantidade de atributos e de relações envolvidas na verificação, mais custosa fica essa verificação;
- RIs quanto ao momento em que deve ser verificada a regra: duas possibilidades podem ser determinadas para essa categoria – verificação imediata ou postergada. A verificação imediata dispara a verificação de todas as RIs associadas a um dado imediatamente após a sua atualização. A verificação postergada realiza a verificação apenas no final da transação ou em algum momento antes do final da transação, quando especificado pelo usuário. A verificação postergada em algumas situações é obrigatória e em outras é recomendada para otimizar o desempenho. Para o controle de verificações postergadas, é fundamental que o SGBD mantenha um histórico de atualizações já feitas (*log*) para poder desfazê-los (*rollback*) em caso de violação;
- RIs quanto aos estados anteriores e posteriores de um dado: essa categoria de RI restringe transições válidas de estado de um dado. Essa transição pode considerar apenas estados anteriores como, por exemplo, o fato de que uma atualização de salário não pode torná-lo inferior ao seu valor antigo, ou anteriores e posteriores como, por exemplo, no caso de uma atualização de estado civil. Nesse segundo exemplo, o SGBD deveria manter um grafo de transições válidas de estado para esse atributo;
- RIs quanto à ocorrência de eventos externos: são RIs que realizam automaticamente ações de atualização, sem que alguma ação desse tipo tenha sido ocasionada por um evento interno (previsível), que é o caso de operações de

atualização. Esse tipo de RI exige que o SGBD constantemente monitore se um evento externo ocorreu, como, por exemplo, um determinado valor de data/hora do sistema.

2.2 SQL E ESPECIFICAÇÃO DE RESTRIÇÕES DE INTEGRIDADE EM SQL

Atualmente, a SQL é a linguagem padrão utilizada para definir e manipular dados em um banco de dados, sendo utilizada em grande parte dos sistemas de banco de dados comerciais.

Cabe esclarecer que a SQL utiliza os termos tabela, linha e coluna para representar uma relação, tupla e atributo, respectivamente. Essa distinção entre SQL e o modelo relacional formal faz-se necessária pelo fato de que a SQL usa o termo tabela para possibilitar a existência de duas ou mais tuplas idênticas, o que no modelo relacional não é permitido.

A SQL permite ao usuário expressar operações para pesquisar e modificar o banco de dados em um modo de alto nível, expressando o que deve ser feito, ao invés de como deve ser feito.

A SQL inclui uma linguagem de definição de dados (DDL) para descrever os esquemas, incluindo restrições de integridade, e uma linguagem para manipulação dos dados (DML) para recuperar e atualizar informações do banco de dados. Alguns exemplos de comandos DDL utilizados são: ALTER, CREATE, DROP e RENAME, que definem dados das relações; CONNECT, GRANT, LOCK e REVOKE, que controlam o acesso aos dados dos esquemas da base de dados. Alguns exemplos de comandos DML utilizados são: DELETE, INSERT e UPDATE, usados para manipular os dados das relações; CLOSE, FETCH, OPEN e SELECT, usados para recuperar os dados das relações; COMMIT, ROLLBACK, SAVEPOINT e SET TRANSACTION, usados para processar as transações das relações da base de dados; DESCRIBE, EXECUTE e PREPARE, usados para aplicação com SQL dinâmico, permitindo criar consultas em tempo de execução (ORACLE, 1999).

A SQL oferece três maneiras de especificar restrições de integridade sobre um esquema relacional (DATE, 2000):

- RIs associadas à definição de relações: dizem respeito a todas as consistências que podem ser feitas sobre um ou mais atributos de uma relação ou sobre atributos de relações associadas a ela. São RIs especificadas no momento da criação de uma relação, estando vinculadas somente a ela e verificadas apenas quando seus atributos são modificados. Nessa situação, encontram-se as RIs de chave primária e chave estrangeira, *not null*, *unique* e todas as RIs enquadradas na categoria "RIs quanto ao alcance". Merece destaque a cláusula *check*, que permite a especificação de diversos tipos de testes comparativos entre atributos, inclusive com a possibilidade de inclusão de comandos de consulta SQL;
- *Asserts*: são testes que devem sempre ter um resultado verdadeiro. Caso contrário, a atualização deve ser desfeita. São empregados geralmente para testes que envolvem operações lógicas ou de cálculo sobre atributos de mais de uma relação, não podendo ser vinculados a uma relação em particular. A SQL permite a sua especificação e remoção do catálogo;
- *Triggers*: importante mecanismo de garantia de integridade semântica, um *trigger* (gatilho) dispara uma ação em função de um evento interno ocorrido no banco de dados (uma operação DML, geralmente), visando manter a integridade do esquema.

2.3 *TRIGGERS* E BANCOS DE DADOS ATIVOS

Um *trigger* é um procedimento que é automaticamente disparado pelo SGBD em resposta a alterações específicas no banco de dados. Um banco de dados que possui um conjunto de *triggers* associados é chamado de banco de dados ativo (ELMASRI; NAVATHE, 2000). Um *trigger* contém três partes:

- Evento: uma alteração no banco de dados que ativa o *trigger*;
- Condição: uma *query* ou teste executado quando o *trigger* é ativado;
- Ação: um procedimento que é executado quando o *trigger* é ativado e a condição for verdadeira.

Pode-se dizer que um *trigger* monitora o banco de dados e é executado quando o banco de dados é modificado, em decorrência de um evento específico. Operações

de inserção, exclusão e atualização podem ativar um *trigger*, sem levar em conta qual o usuário ou aplicação que executou a operação.

Uma condição em um *trigger* pode ser uma afirmação verdadeira/falsa ou uma *query*. Uma *query* é interpretada como verdadeira se o conjunto de resposta for não vazio e/ou falsa se a *query* não retornar respostas. Se a condição avaliada for verdadeira, a ação associada com o *trigger* é executada.

A ação do *trigger* pode examinar as respostas da *query*, referenciando-se aos novos e velhos valores das tuplas modificadas pela operação que ativou o *trigger*, executar novas *queries* e realizar alterações no banco de dados.

Triggers podem auxiliar no controle de acessos e privilégios, na melhoria da integridade dos dados, no reforço a complexas restrições de integridade referencial, na auditoria sobre a manipulação dos dados e na replicação de dados (ELMASRI; NAVATHE, 2000).

Destacamos o uso freqüente de *triggers* para manter a consistência do banco de dados, uma vez que seu uso permite manter a integridade no banco de dados de maneira mais flexível. Porém, em muitos casos, são difíceis de serem compreendidos e deve-se considerar o uso de outro recurso para manter a consistência. Um conjunto de *triggers* em um banco de dados pode ser muito complexo, e a manutenção desse banco de dados ativo pode ser muito difícil.

Em um sistema de banco de dados ativo, quando o SGBD é chamado a executar um procedimento que modifica o banco de dados, ele verifica se algum *trigger* está ativado para esse procedimento. Se existir, o SGBD processa o *trigger* avaliando a parte da condição e, se a condição avaliada for verdadeira, executa a parte da ação.

Se um procedimento ativa mais de um *trigger*, o SGBD processa todos eles, em alguma ordem arbitrária. Um ponto importante a ser considerado é que a execução de um *trigger* pode, por sua vez, ativar outro *trigger*. A execução da parte da ação de um *trigger* pode, de novo, ativar o mesmo *trigger*. Esses *triggers* são chamados *triggers* recursivos. Esse tipo de ativação em série e a imprevisível ordem em que o SGBD processa os *triggers* ativados podem tornar difícil a compreensão dos efeitos de

um conjunto de *triggers* (ELMASRI; NAVATHE, 2000).

2.4 PROJETO DE BANCO DE DADOS

O desenvolvimento de um sistema de informação, segundo PRESSMAN (1998), envolve a análise e o projeto de dois componentes: dados e processos. O projeto de dados é considerado a parte estática do sistema, uma vez que diz respeito a um universo persistente de características que dificilmente sofre modificações após a sua definição. O projeto de processos, por sua vez, é chamado de parte dinâmica, uma vez que as tarefas a serem realizadas sobre os dados podem variar, conforme ocorre a evolução do sistema.

Fazem parte de um projeto de um banco de dados a análise, o projeto e a implementação dos dados persistentes de uma aplicação, levando-se em conta a determinação da sua semântica (abstração dos dados de uma realidade) e posteriormente, o modelo de dados e o SGBD a serem adotados.

Segundo ELMASRI e NAVATHE (2000), o projeto de um banco de dados é composto, basicamente, de quatro etapas: descrição de requisitos, projeto conceitual, projeto lógico e projeto físico.

2.4.1 Descrição de Requisitos

Na etapa de descrição de requisitos são coletadas informações sobre os dados de interesse da aplicação, o seu uso (operações de manipulação sobre eles) e suas relações. O resultado dessa etapa normalmente é uma descrição dos requisitos da aplicação, de acordo com a metodologia de engenharia de *software* que esteja sendo empregada.

2.4.2 Projeto Conceitual

Essa etapa pode ser considerada a fase de análise dos dados (ou requisitos) capturados na etapa anterior. Nessa etapa é realizada a chamada *modelagem conceitual*: são analisados os fatos (entidades ou conjunto de ocorrências de dados) de interesse e seus relacionamentos, juntamente com seus atributos (propriedades ou características) e é construída uma notação gráfica (abstrata, uma representação de alto nível) para facilitar o entendimento dos dados e suas relações, tanto para os analistas quanto para os futuros usuários. Essa etapa resulta em um modelo conceitual, no qual a semântica da realidade deve estar correta. Um recurso muito empregado nessa etapa é o Diagrama Entidade-Relacionamento (DER).

2.4.3 Projeto Lógico

É considerado a fase de projeto dos dados. Nessa etapa, o desenvolvimento do banco de dados começa a se voltar para o ambiente de implementação, uma vez que é feita a conversão do modelo conceitual para um modelo de dados de um banco de dados (modelo lógico). Este modelo de dados pode ser o modelo relacional, orientado a objetos, etc. Essa etapa se baseia no uso de regras de mapeamento do DER para o modelo de dados escolhido. No caso do modelo relacional, o resultado é uma estrutura lógica, como um conjunto de tabelas relacionadas.

2.4.4 Projeto Físico

Essa última etapa realiza a adequação do modelo lógico gerado na etapa anterior ao formato de representação de dados do SGBD escolhido para a implementação. Para a realização dessa etapa, deve-se conhecer a DDL e a DCL do SGBD, para realizar a descrição do modelo lógico. O resultado é a especificação do esquema da aplicação, juntamente com a implementação de restrições de integridade.

2.4.5 Problemas em Projeto de Banco de Dados

O projeto de banco de dados é um processo complexo, uma vez que o projetista tem de considerar não apenas como modelar o mundo real, mas também as limitações do sistema de banco de dados e a eficiência da recuperação e da atualização dos dados (CHEN, 1992).

Apesar dos muitos esforços (métodos e ferramentas) no sentido de auxiliar o processo de projeto de banco de dados, o projetista ainda tem de contar, muitas vezes, com sua intuição e experiência. Como resultado, muitos bancos de dados são projetados de forma inadequada.

Dentre as principais causas de erros em projetos de banco de dados, podemos destacar:

- Não utilização de métodos e ferramentas;
- Adaptações no projeto original como conseqüência de, por exemplo, restrições impostas pelos tipos limitados de estrutura de dados que são suportados pelo SGBD, querer tornar a recuperação e a atualização mais eficientes ou conseguir uma melhor utilização do espaço em disco;
- Inclusão de defeitos no projeto, durante o mapeamento objeto-relacional, quando o processo de desenvolvimento utiliza linguagem de programação orientada a objetos e SGBD relacional;
- Utilização descontrolada de um conjunto de *triggers*.

2.5 DIAGRAMA ENTIDADE-RELACIONAMENTO (DER)

O Diagrama Entidade-Relacionamento (DER), amplamente utilizado nos projetos de banco de dados, representa a modelagem conceitual de banco de dados, sendo considerado praticamente um padrão. É de fácil compreensão e apresenta poucos conceitos. Segundo CHEN (1992), um DER é uma representação gráfica, na forma de um diagrama, na qual são utilizados apenas três conceitos: entidade, relacionamento e atributo.

2.5.1 Entidade

Entidades representam fatos do mundo real, sejam eles concretos ou abstratos, sobre os quais se deseja manter dados em um banco de dados. São representados por retângulos nomeados (substantivos) que indicam um conjunto de ocorrências da entidade, conforme exemplificado na Figura 1. Sugere-se que sejam utilizados nomes no singular. São exemplos de entidades: empregado, departamento, aluno, curso, etc.

FIGURA 1 – EXEMPLO DE NOTAÇÃO DE ENTIDADES EM UM DER.

2.5.2 Relacionamento

Relacionamentos representam associações entre entidades. Um relacionamento indica uma função cumprida pelas entidades envolvidas, sendo representado por um losango. Sugere-se que o nome seja um substantivo no singular. Em cada associação são indicadas as cardinalidades, ou seja, o número de ocorrências de uma entidade que se relacionam com uma ocorrência de outra entidade. Quando não se sabe exatamente o número de ocorrências, representa-se com a letra "N". Um exemplo de relacionamento está representado na Figura 2.

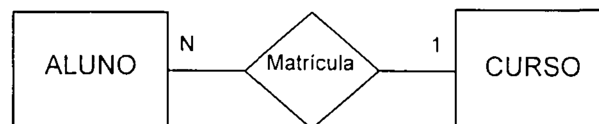


FIGURA 2 – EXEMPLO DE NOTAÇÃO DE RELACIONAMENTO EM UM DER.

Os relacionamentos podem ser binários, ternários, etc, dependendo da quantidade de entidades diferentes envolvidas. Um relacionamento reflexivo é um tipo de associação que envolve ocorrências em uma mesma entidade.

2.5.3 Atributo

Atributos, como já apresentado anteriormente, representam uma propriedade de uma entidade ou de um relacionamento como, por exemplo, o salário de um empregado ou o tempo que um empregado estará alocado a um projeto.

Existem alguns tipos especiais de atributos:

- Atributos opcionais: propriedades que podem assumir *null*;
- Atributos compostos: representam uma abstração de outros atributos como, por exemplo, um endereço, que abstrai os dados rua, número, cidade, CEP, etc;
- Atributos multivalorados: propriedades que podem assumir mais de um valor como, por exemplo, os números de telefone de um departamento.

2.6 MAPEAMENTO OBJETO-RELACIONAL

Apesar de existirem há muitos anos, os sistemas gerenciadores de banco de dados orientados a objeto (SGBD-OO) são pouco utilizados em aplicações convencionais, ao contrário das linguagens de programação orientadas a objeto, que são largamente difundidas e utilizadas. A pouca força dos SGBD-OO contrasta com a grande força e o grande uso dos bancos de dados relacionais. Isto talvez se deva à robustez e ao desempenho encontrados no modelo relacional ou ao fato de que os bancos relacionais já estão consolidados há algum tempo.

Atualmente, na maior parte dos sistemas desenvolvidos pelo paradigma da orientação a objeto, a linguagem de programação utilizada é orientada a objeto, porém, o sistema de gerenciamento de banco de dados é relacional. Nesses sistemas, torna-se necessário dispor de mecanismos que mapeiem objetos em memória para meios persistentes relacionais. Este processo, chamado de mapeamento objeto-relacional, não é imediato e, atualmente, as linguagens orientadas a objeto não fornecem suporte direto para o mesmo.

No mapeamento objeto-relacional, todo objeto a ser armazenado no repositório deve ser mapeado para uma representação no modelo relacional como, por

exemplo, um objeto de uma determinada classe que é mapeado em uma tupla da tabela que representa sua classe.

O mapeamento objeto-relacional pode ser realizado tanto manualmente quanto automaticamente. O processo automático, suportado por uma ferramenta, tem se mostrado mais simples e mais rápido do que o manual. Porém, o processo automático nem sempre é transparente e o custo de uma ferramenta para essa atividade ainda é bastante elevado.

A preocupação desse trabalho é que, durante o mapeamento objeto-relacional, erros possam ser incluídos no projeto do banco de dados, principalmente quando ele é realizado manualmente.

2.7 CONSIDERAÇÕES FINAIS

Neste capítulo, são apresentados alguns conceitos básicos e terminologia de Banco de Dados Relacionais relacionados com projeto de banco de dados e com integridade e consistência dos dados.

No próximo capítulo, são apresentados alguns conceitos básicos e terminologia de Teste, Validação e Verificação (V&V).

3 TESTE E VALIDAÇÃO: CONCEITOS E TERMINOLOGIA

3.1 A ATIVIDADE DE TESTE E O TESTE DE BASES DE DADOS

Durante o processo de desenvolvimento de um produto de *software*, falhas, freqüentemente humanas, podem ser cometidas, ocasionando a introdução de defeitos nesse produto. Assim, técnicas e métodos devem ser utilizados no sentido de tornar esse processo o mais confiável possível.

Uma das atividades que se tem mostrado essencial para o aumento da qualidade e confiabilidade de um produto é o teste, atividade importante na avaliação de qualquer produto de *software* (PRESSMAN, 1998). O teste é um processo cuja finalidade é detectar defeitos em um produto e pode ser usado, também, para avaliar o quanto o produto atende às necessidades especificadas (MYERS, 1979).

Tradicionalmente, o teste de programas tem sido o foco principal do esforço de pesquisa e desenvolvimento. O teste de programas consiste, basicamente, em executar um programa fornecendo dados de entrada e comparar a saída alcançada com o resultado esperado, obtido na especificação do programa (MYERS, 1979). Mais informações sobre esse tipo de teste podem ser encontradas no Anexo 2 deste trabalho.

No teste de base de dados, a estrutura lógica dos dados passa a ser o objeto do teste e o processo consiste em executar operações sobre uma base de dados com o objetivo de verificar se há ocorrência de falhas (ARANHA et al., 2000). A idéia, no teste de base de dados, é exercitar as necessidades especificadas, de maneira que o banco de dados possa atendê-las com segurança, para futuramente contemplar todas as funcionalidades de suas aplicações.

O teste de base de dados representa uma importante chance de se detectar defeitos introduzidos no projeto, antes que a base de dados seja povoada e utilizada efetivamente. Assim, verifica-se a necessidade do desenvolvimento e do aprimoramento de métodos, técnicas e ferramentas para apoiar a atividade de teste de bases de dados. Cumpre observar que este trabalho concentra-se em teste de bases de dados relacionais.

3.1.1 Conceitos em Teste de Bases de Dados

O texto desta seção, baseado em (ARANHA et al., 2000), apresenta alguns conceitos e considerações sobre teste de bases de dados relacionais.

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

Um caso de teste consiste na associação de uma *query* com o resultado esperado. Um exemplo de resultado esperado são as possíveis tuplas afetadas e os respectivos valores.

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

Critérios de teste estabelecem requisitos a serem satisfeitos pela execução do teste e podem auxiliar na seleção de dados de teste e na avaliação da qualidade de teste. Satisfazer um critério significa atender aos requisitos de teste estabelecidos pelo critério. Elementos requeridos de um critério são os elementos da base de dados que devem ser exercitados para que se o considere satisfeito. Cobertura de um critério é uma medida do percentual dos elementos requeridos que foram efetivamente exercitados em relação ao número total de elementos requeridos pelo critério. É uma medida de avaliação da qualidade de um teste segundo um critério.

3.1.2 O Estado do Banco de Dados Durante o Teste

Em um sistema de banco de dados relacional, uma relação descreve o estado dos dados no momento atual. O estado da relação muda toda vez que houver uma inclusão, exclusão ou alteração no banco de dados, refletindo mudanças do mundo real.

Quando se lida com teste de bases de dados deve-se estar atento ao estado do

banco de dados antes e depois da execução de um caso de teste (CHAYS et al., 2000). Isso se faz necessário, uma vez que uma falha, causada por um caso de teste anterior, pode, por exemplo, modificar o estado da relação de forma a tornar o banco de dados inconsistente, influenciando no resultado do caso de teste atual. Caso necessário, deve-se colocar o banco de dados no estado desejado antes da execução de um caso de teste, e observar o seu estado depois da execução do caso de teste.

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

Observar o estado do banco de dados após a execução dos casos de teste pode envolver várias técnicas como, por exemplo, extrair informações dos arquivos de históricos ou comparar dados das tabelas antes e depois da execução dos casos de teste.

3.2 AS ATIVIDADES DE VERIFICAÇÃO E VALIDAÇÃO (V&V)

Verificação e validação (V&V) compreendem uma série de atividades técnicas e de gerenciamento desenvolvidas para melhorar a qualidade e a confiabilidade de um sistema e garantir que o produto entregue satisfaça as necessidades operacionais do cliente ou usuário (LEWIS, 1992).

V&V abrangem um conjunto amplo de atividades que incluem revisões técnicas formais, auditorias de configuração e qualidade, monitoração do desempenho, simulação, estudo da viabilidade, revisão da documentação, revisão de bancos de dados, análise de algoritmos, teste de desenvolvimento, teste de qualificação e teste de instalação.

PRESSMAN (1998) faz uma diferenciação entre verificação e validação. A meta da verificação é assegurar que a equipe de desenvolvimento construa corretamente o sistema, isto é, assegurar que cada fase do processo de desenvolvimento do sistema seja construída de acordo com as especificações definidas previamente para a fase. A meta da validação é assegurar que a equipe de

desenvolvimento construa o sistema certo, isto é, assegurar que o sistema seja construído de acordo com as necessidades do usuário.

Encontramos verificação nas fases de análise de requisitos, projeto e codificação/implementação do processo de desenvolvimento. Validação, por outro lado, testa o sistema para garantir que os requisitos foram atendidos. Segundo LEWIS (1992), as fases de um processo de V&V são: verificação de requisitos, verificação de projeto, verificação do código e validação.

3.2.1 Verificação de Requisitos

A fase de verificação de requisitos, segundo LEWIS (1992), tem como meta principal garantir que os requisitos sejam expressos de forma correta e completa na especificação do sistema e que eles sejam decompostos e alocados aos requisitos de *hardware* e *software*. Uma vez que todos os requisitos tenham sido apontados, V&V iniciam a engenharia de avaliação e atividades de verificação para examinar a corretitude, a consistência e a testabilidade. Virtualmente, todos os requisitos podem ser testados de alguma maneira, mas em alguns casos pode não ser muito simples. Entretanto, deve existir uma estratégia para checar cada um deles. Simulação e modelagem são ferramentas que podem ser utilizadas pela verificação para confirmar o comportamento e/ou desempenho.

Um dos objetivos da verificação de requisitos é tentar descobrir algo que o desenvolvedor não tenha percebido ou não tenha entendido completamente. Na engenharia de requisitos há duas técnicas que podem contribuir significativamente para o processo de análise de requisitos: prototipação rápida e linguagens formais de especificação.

Dentre as atividades desenvolvidas pela verificação de requisitos (LEWIS, 1992), podemos destacar: verificação da especificação do sistema; verificação da especificação de requisitos do *software*; verificação da especificação dos requisitos de interface; verificação do plano de desenvolvimento de *software* e sua implementação; revisão das necessidades de interface do usuário; revisão dos requisitos dos dados de

entrada e saída; revisão dos requisitos, dos planos e das estratégias de teste; participação em todas as revisões e encontros que afetem o sistema; produção de um relatório de problemas.

3.2.2 Verificação de Projeto

A fase de verificação de projeto, segundo LEWIS (1992), pode ser considerada como um processo de duas partes: projeto preliminar e projeto detalhado.

O projeto preliminar ajuda a identificar as interfaces, os dados e o fluxo de controle entre eles (estados, memória e tempo necessários, etc). Na maioria dos casos, V&V verificam todos os requisitos com referência cruzada entre a especificação de requisitos do *software* e o documento de projeto do *software*. A modelagem e a simulação podem ser utilizadas para avaliar o desempenho, caso necessário. Uma das dificuldades é saber onde parar com o processo de verificação do projeto preliminar. É importante saber que, nesse nível, alguns detalhes são desconsiderados, mas a estrutura e o comportamento dos componentes devem ser identificados. Informações relacionadas à seqüência de operações e ao fluxo dos dados, obtidas, por exemplo, por meio de prototipação e simulação, podem ser usadas para equilibrar o processo em relação ao tempo e aos recursos disponíveis. Isso é o mais longe que a verificação do projeto preliminar deve ir. Antes de se seguir adiante, para o projeto detalhado, deve-se ter a certeza de que o projeto está completo e correto o suficiente.

O projeto detalhado aborda o nível lógico, caracterizado pelos arquivos de dados, registros, índices, lógica de programação, algoritmos, etc. V&V descem ao nível da unidade. É como se o projeto preliminar fosse "explodido" para apresentar todos os detalhes do projeto, para que sejam avaliados. V&V podem ser iniciadas tão logo os primeiros módulos sejam escritos a fim de que seja possível um *feedback* o quanto antes.

Dentre as atividades desenvolvidas pela verificação de projeto, segundo LEWIS (1992), podemos destacar: verificação do documento de projeto do *software*; verificação do documento de projeto da interface; verificação do plano de teste do

software para essa fase; verificação de alguns algoritmos (começando pelos mais críticos); revisão da observância do desenvolvedor ao plano de desenvolvimento do *software*; participação em todas as revisões e encontros que afetem o sistema; produção de um relatório de problemas.

3.2.3 Verificação de Código

A fase de verificação de código, segundo LEWIS (1992), compreende a avaliação da consistência entre o código e o documento de projeto de *software*, e se o código segue os padrões e o plano de desenvolvimento do *software*. Isso envolve a avaliação de porção de código, lógica, estruturas de dados, caminhos de execução, interfaces, reutilização de código, etc.

Os principais motivos de realizar a verificação de código, segundo LEWIS (1992), são: detectar erros antes de o código passar do ambiente de programação para o ambiente de teste; garantir que um bom projeto e práticas de código tenham sido seguidos e; garantir que o projeto do *software* está corretamente representado no código.

Dentre as atividades desenvolvidas pela verificação de código, segundo LEWIS (1992), podemos destacar: verificação da consistência entre o código e o documento de projeto do *software*; verificação da observância aos padrões; verificação da estrutura lógica e da sintaxe; verificação dos termos no dicionário de dados; verificação dos dados de entrada e saída; revisão da biblioteca de *software* e controles de versão/*release*; participação em todas as revisões e encontros que afetem o sistema; produção de um relatório de problemas.

3.2.4 Validação

Validação, segundo LEWIS (1992), é a atividade de V&V que faz que os critérios de validação estabelecidos durante a análise de requisitos sejam testados. O teste de validação, seguindo um plano de teste, oferece a garantia final de que o

software ou sistema atende a todas as exigências funcionais, comportamentais e de desempenho.

Cabe salientar que, embora o plano de teste tenha sido produzido durante o projeto preliminar e a descrição do teste seja produzida no projeto detalhado, sua revisão e avaliação são atividades da validação e não do projeto.

A validação inicia-se juntamente com a fase de integração e teste e continua até o término do teste do sistema. Ela não só confirma os requisitos, por meio do teste, como também garante que as especificações do produto e os documentos do usuário e operador estejam livres de erro, representando o sistema atualmente construído.

Dentre as atividades desenvolvidas pela validação, segundo LEWIS (1992), podemos destacar: revisão e avaliação do plano de teste; validação do teste relativo a cada requisito; realização de uma avaliação de interface; execução de simulações nos algoritmos (principalmente os mais críticos) e comparação do desempenho atual com os resultados obtidos nas simulações anteriores; validação dos manuais de usuário e de operação; participação nas revisões, auditorias e encontros que afetem o sistema; produção de um relatório de problemas.

3.3 CONSIDERAÇÕES FINAIS

Neste capítulo, são apresentados alguns conceitos básicos e terminologia de Teste, Validação e Verificação (V&V). Esses conceitos, juntamente com os conceitos de Banco de Dados Relacional, formam uma base para o desenvolvimento deste trabalho.

No próximo capítulo, são apresentados o modelo de processo de validação de projeto proposto e a Ferramenta DBValTool.

4 O MODELO PROPOSTO E A FERRAMENTA DBValTool

O modelo de processo de validação de projeto e a Ferramenta DBValTool (*DataBase Testing and Validation Tool*), apresentados neste trabalho, representam esforços no sentido de apoiar o teste de bases de dados e a validação de projeto, a fim de assegurar que este seja implementado da forma como a sua especificação determina.

O modelo proposto permite avaliar o quanto estão corretas e completas as definições criadas no banco de dados, visando detectar possíveis erros na criação dos esquemas relacionais, antes de passar para etapas seguintes do processo de desenvolvimento.

A DBValTool apóia o teste de bases de dados, buscando proporcionar um aumento da qualidade e uma redução no tempo e no custo da atividade de teste. A Ferramenta apóia o processo de validação de projeto, podendo ser utilizada em várias etapas do modelo proposto.

4.1 PROPOSTA DE MODELO DE PROCESSO DE VALIDAÇÃO DE PROJETO

O modelo proposto apresenta uma abordagem ao processo de validação de projeto de banco de dados relacional, com o objetivo principal de aumentar a qualidade e a confiabilidade do projeto. Outro objetivo do modelo é contribuir para conscientização e consolidação da necessidade desse tipo de processo.

A fim de alcançar os seus objetivos, o modelo propõe um conjunto sistemático e planejado de atividades, sendo estas organizadas em etapas. Idealmente, cada etapa do processo deve produzir um resultado que possa ser revisado e possa funcionar como uma base para as etapas que se seguirão. O projeto validado é produzido ao término das etapas.

O modelo de processo de validação de projeto proposto, representado na Figura 3, é composto pelas etapas descritas a seguir:

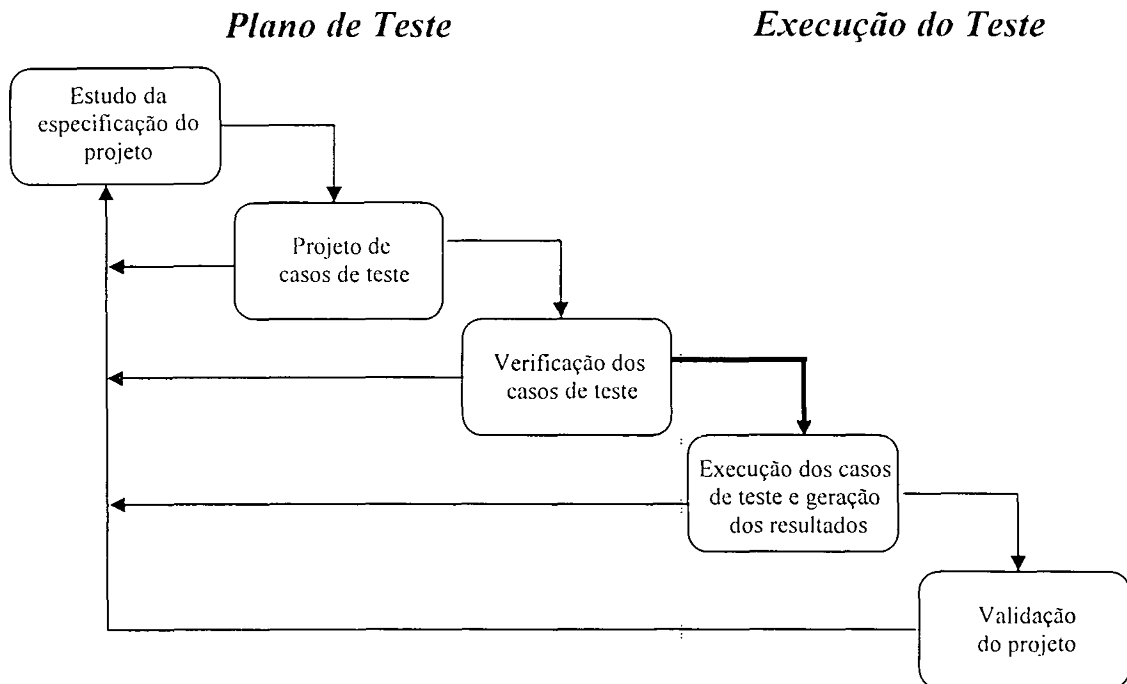


FIGURA 3 - MODELO DE PROCESSO DE VALIDAÇÃO DE PROJETO

Etapa 1: Estudo da especificação do projeto

Inicialmente, o testador deve estudar a especificação do projeto de forma que o conhecimento do projeto seja garantido. A especificação do projeto é o documento que serve de base para a elaboração do plano de teste. Ela descreve a arquitetura das bases de dados do sistema e as restrições necessárias para garantir a integridade dos dados, proporcionando ao testador condições para avaliar a qualidade logo que o projeto for testado. Nessa fase o testador deve tomar conhecimento de todas as características e restrições do projeto a ser validado. Uma compreensão completa da especificação do projeto é fundamental para um bem-sucedido projeto de casos de teste.

Etapa 2: Projeto dos casos de teste

Essa etapa traduz um conjunto de informações contidas na especificação do projeto em casos de teste. Em teste de bases de dados, um caso de teste consiste na associação de uma *query* com o resultado esperado. O testador deve definir e elaborar

os casos de testes a serem utilizados no processo, com base nas informações contidas na especificação do projeto e na sua experiência. Os casos de teste devem evidenciar a maior quantidade de erros possíveis, com uma quantidade mínima de esforço e tempo. Os casos de testes devem ser elaborados no sentido de verificar se as restrições estão realmente sendo implementadas, e se o estão fazendo de forma correta. O uso ou não de critérios de teste fica sob responsabilidade do testador, porém o seu uso é desejável, uma vez que proporcionam uma maneira sistemática de testar o esquema da base de dados e auxiliam na avaliação da qualidade do teste.

Etapa 3: Verificação dos casos de teste

Verifica-se, por meio de uma análise detalhada, a abrangência e a consistência dos casos de testes elaborados na etapa anterior. Julga-se necessária essa etapa uma vez que, como em toda a engenharia de sistemas, há uma tendência de passar-se rapidamente para a etapa seguinte. Nessa etapa, faz-se uma revisão dos casos de teste elaborados, a fim de garantir que o escopo do teste foi corretamente projetado e os casos de teste são em quantidade e qualidade suficientes para realizar os testes dentro de um esforço e tempo compatíveis com o disponível. Caso necessário, o testador pode eliminar ou aperfeiçoar os casos de teste existentes, bem como elaborar novos casos.

Etapa 4: Execução dos casos de teste e geração dos resultados

A execução dos casos de testes consiste em executar operações, elaboradas nas etapas anteriores, com o objetivo de exercitar os esquemas relacionais.

Cabe lembrar que, quando se lida com teste de bases de dados, deve-se estar atento ao estado do banco de dados antes e depois da execução do caso de teste, uma vez que uma falha ocasionada por um caso de teste anterior pode influenciar no resultado do caso de teste atual. Caso necessário, deve-se colocar o banco de dados no estado desejado antes da execução de um caso de teste e observar seu estado depois da execução.

Os resultados gerados devem compreender o máximo de informações possíveis sobre a execução das operações como, por exemplo, mensagens e códigos de

retorno do banco de dados, conteúdo de *triggers* e *procedures* executadas, possíveis modificações realizadas nos dados, etc. Os resultados gerados poderão apoiar a validação do projeto.

Etapa 5: Validação do projeto

Para a validação do projeto ter êxito, é necessário avaliar os resultados obtidos na etapa anterior e compará-los com a especificação do projeto. À medida que os resultados são avaliados, uma indicação da qualidade e da confiabilidade pode ser observada. Se resultados indesejados começarem a aparecer, a qualidade e a confiabilidade do projeto são suspeitas e, provavelmente, defeito(s) na implementação do projeto foi(ram) encontrado(s). Caso necessário, após corrigido(s) o(s) defeito(s), as Etapas 4 e 5 podem ser novamente processadas, para os casos de teste que detectaram o(s) defeito(s). Após avaliar a execução de todos os casos de teste definidos, o testador pode, se julgar necessário, aperfeiçoar os casos de teste existentes e/ou elaborar novos casos. A validação termina quando os resultados obtidos forem considerados satisfatórios e suficientes para garantir a qualidade do projeto. Alerta-se para que não haja um negligenciamento dessa etapa, uma vez que, para a sua conclusão, exige-se um minucioso entendimento do projeto e a dedicação de muito tempo e atenção.

4.2 A FERRAMENTA DBValTool

Assim como nas outras áreas, a área de teste vem se utilizando cada vez mais de ferramentas automatizadas para apoiar as suas atividades. As ferramentas de teste, em sua maioria, reduzem o tempo de teste e aumentam a sua eficácia (PRESSMAN, 1998).

A Ferramenta DBValTool foi desenvolvida com o objetivo principal de proporcionar um aumento da qualidade e uma redução no tempo e no custo da atividade de teste de bases de dados relacionais. Para isso, suporta uma abordagem onde os testes são realizados sobre relações e relacionamentos, e os elementos a serem exercitados, por meio de *queries*, são os atributos do esquema de uma base de dados

relacional.

A Ferramenta pode apoiar, também, o processo de validação de projeto. Para tanto, ela oferece ao testador um ambiente de teste – gráfico, simples e integrado com o banco de dados –, com funcionalidades que podem ser utilizadas nas várias etapas do processo de validação, buscando proporcionar um aumento da qualidade e da confiabilidade do projeto.

A DBValTool foi desenvolvida pelo paradigma da orientação a objeto e o processo utilizado no seu desenvolvimento foi o RUP (*Rational Unified Process*), apoiado pela ferramenta Rational Rose®, utilizando-se a notação UML. Mais informações sobre UML podem ser encontradas no Anexo 1 deste trabalho.

4.2.1 Aspectos Funcionais

O diagrama de *use cases*, conforme a Figura 4, apresenta os casos de uso da Ferramenta DBValTool por um testador.

A DBValTool implementa as seguintes funcionalidades:

- Criação de sessões de teste, segundo o plano de teste e/ou necessidades do testador, para a execução dos casos de teste, que consistem na associação de uma *query* com o resultado esperado;
- Entrada de operações de consultas e atualizações da base de dados, escritas pelo testador ou criadas por um assistente, no formato e padrão SQL;
- Apresentação dos resultados da execução das operações de consultas e atualizações da base de dados, envolvendo informações sobre, por exemplo, relações e tuplas afetadas, restrições de integridade exercitadas, *triggers* disparadas, *procedures/functions* executadas, etc;
- Consulta a informações contidas no dicionário do banco de dados;
- Extração, a partir do esquema da base de dados, de informações sobre definições e declarações formais como, por exemplo, tabelas, atributos, tipos, tamanhos, restrições de domínio, restrições de integridade referencial, etc;

- Avaliação estática e dinâmica da base de dados em termos, respectivamente, das definições e declarações formais e dos eventos ocorridos durante os testes, com o objetivo de detectar possíveis defeitos na especificação do seu projeto.

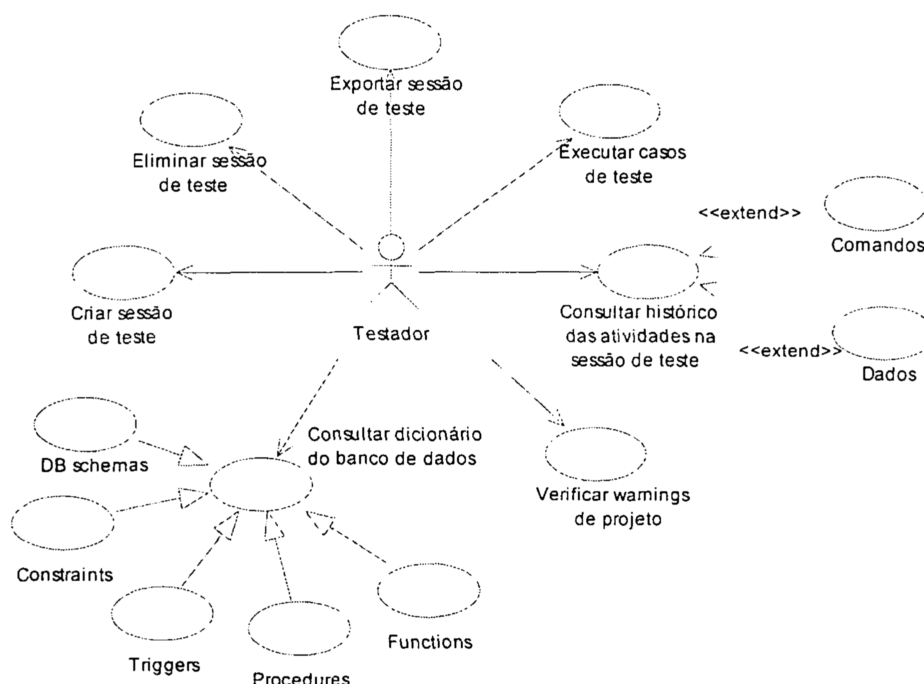


FIGURA 4 – DIAGRAMA DE *USE CASES* DA DBValTool

4.2.2 Arquitetura

A arquitetura da DBValTool, representada na Figura 5, ilustra três camadas: interface, base e dados.

Na camada superior temos o módulo de interface com o testador. Através dessa camada todas as interações com a Ferramenta serão realizadas. Para o testador, essa é a única camada que ele tem acesso, as duas restantes são transparentes.

A camada base é composta pelos módulos responsáveis pelo processamento das informações. Por um lado, temos o processamento das informações fornecidas pelo testador e, por outro, o tratamento das respostas às operações no banco de dados.

Na camada inferior temos a base de dados, responsável pelo armazenamento das informações das sessões de teste como, por exemplo, histórico de comandos, histórico de dados e informações relativas às configurações.

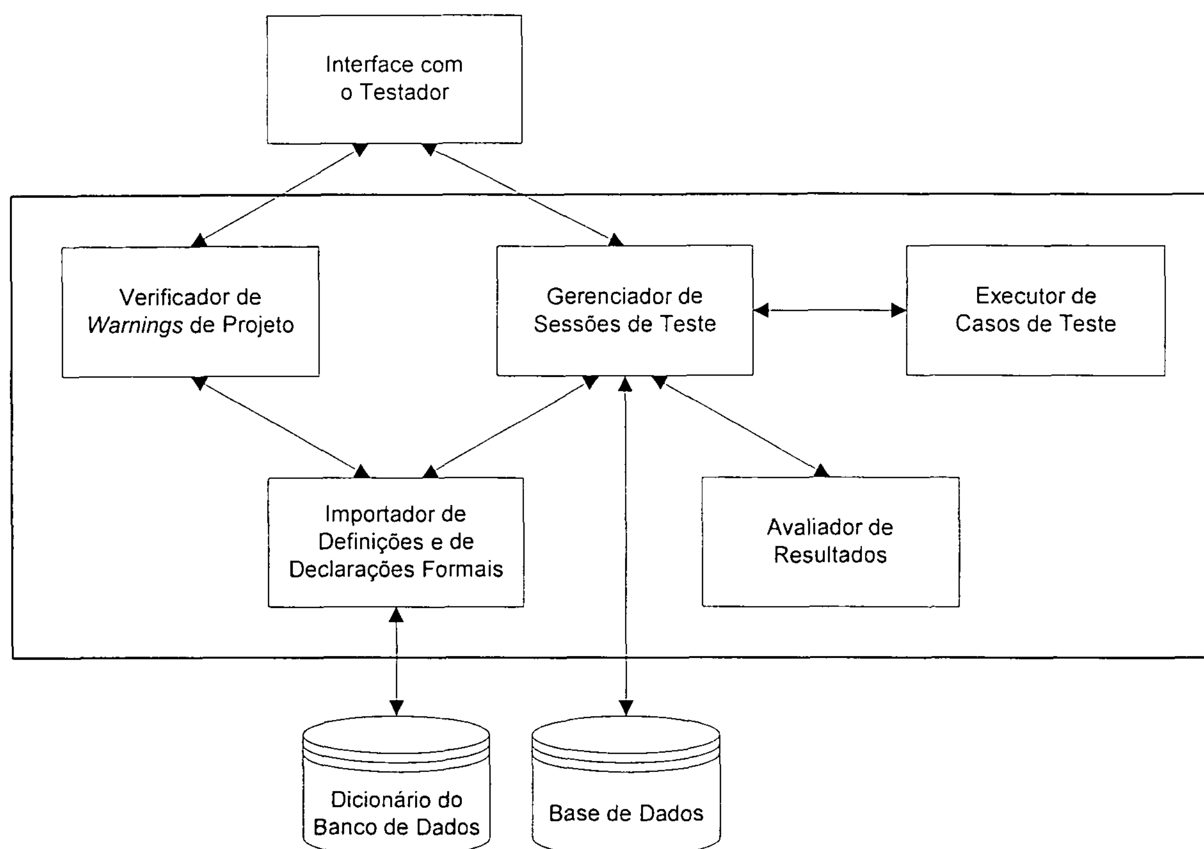


FIGURA 5 – ARQUITETURA DA DBValTool

Os módulos da DBValTool estão descritos a seguir:

- **Interface com o testador**: é responsável pela comunicação entre o testador e a Ferramenta e pela seqüenciação das atividades de teste, por meio da ativação dos demais módulos.
- **Gerenciador de Sessões de Teste:** controla as sessões de teste, criadas e usadas pelo testador para a execução dos casos de teste e para a avaliação dos resultados desses testes. O critério para criar uma sessão deve ser definido pelo testador e poderá ser determinado, por exemplo, pelo tempo necessário à execução dos testes ou pela funcionalidade a ser testada. Se desejar, o testador poderá criar uma ou várias sessões durante os testes.

Uma sessão de teste contém informações de identificação da sessão (nome da sessão, nome do testador, projeto em teste, nome da empresa, etc), bem como informações relacionadas à execução dos casos de teste (conexões com o banco de dados, histórico das operações executadas, histórico dos dados alterados, etc).

- **Executor de Casos de Teste:** controla a execução dos casos de teste. Possui como entrada uma *query*, construída por um assistente ou digitada pelo testador, e, como saída, os resultados da execução dessa *query*. Os resultados gerados compreendem informações que podem ser utilizadas pelo testador para avaliar a qualidade dos testes e apoiar a validação do projeto. Essas informações são, por exemplo, mensagens e códigos de retorno do banco de dados, *triggers* disparadas, possíveis alterações realizadas nos dados, etc.
- **Avaliador dos Resultados:** apóia a avaliação dos resultados por meio da disponibilização do histórico de operações executadas e do histórico de dados alterados, da consulta ao dicionário do banco de dados e da geração de relatórios das atividades realizadas na sessão de teste.
- **Verificador de *Warnings* de Projeto:** gera um relatório de advertências, relacionadas ao projeto da base de dados, que devem ser analisadas pelo testador, uma vez que podem ser ou vir a se tornar um defeito causador de erros. Essas advertências são resultantes de uma análise das informações extraídas do esquema da base de dados atual e do histórico dos defeitos em projetos de bases de dados mais observados pelos testadores. Inicialmente, sete advertências foram especificadas e implementadas na Ferramenta. Um exemplo de advertência implementada é a declaração de chave estrangeira sem indicação da ação (*restrict*, *delete*, *cascade*, *set null*) a ser adotada nos casos de exclusão ou atualização da chave primária referenciada.
- **Importador de Definições e Declarações Formais:** extrai do dicionário do banco de dados informações sobre definições e declarações formais

como, por exemplo, tabelas, atributos, tipos, tamanhos, restrições de domínio, restrições de integridade referencial, etc.

O modelo da base de dados para o armazenamento das informações das sessões de teste pode ser visto na Figura 7.

4.2.3 Implementação

A DBValTool foi implementada em Borland® Delphi para uso em ambiente Windows®, podendo ser distribuída, também, para o ambiente Linux por meio do Borland® Kylix. A escolha por esse ambiente de desenvolvimento deu-se, principalmente, pela disponibilidade em nossa instituição, pela relativa portabilidade e pela facilidade de oferecer acesso a diversos SGBDs.

Na sua primeira versão, a Ferramenta oferece um ambiente de testes para projetos implementados no banco de dados relacional Oracle®. Escolhemos esse banco de dados devido à grande base instalada, por ser muito utilizado comercialmente e pela disponibilidade em nossa instituição. A comunicação entre a Ferramenta e o SGBD é feita por roteador direto (sem o uso de ODBC – *Open Database Connectivity*), o que a torna mais estável e veloz. A idéia é de, futuramente, ampliar a lista de bancos de dados suportados pela Ferramenta.

A DBValTool foi desenvolvida pelo paradigma da orientação a objeto e o seu diagrama de classes, elaborado durante o processo de desenvolvimento da Ferramenta, está apresentado na Figura 6.

Como foi utilizado um sistema gerenciador de banco de dados relacional para o armazenamento dos dados, tornou-se necessário realizar o mapeamento objeto-relacional, por meio do qual os objetos em memória são mapeados para meios persistentes relacionais, de forma transparente ao usuário.

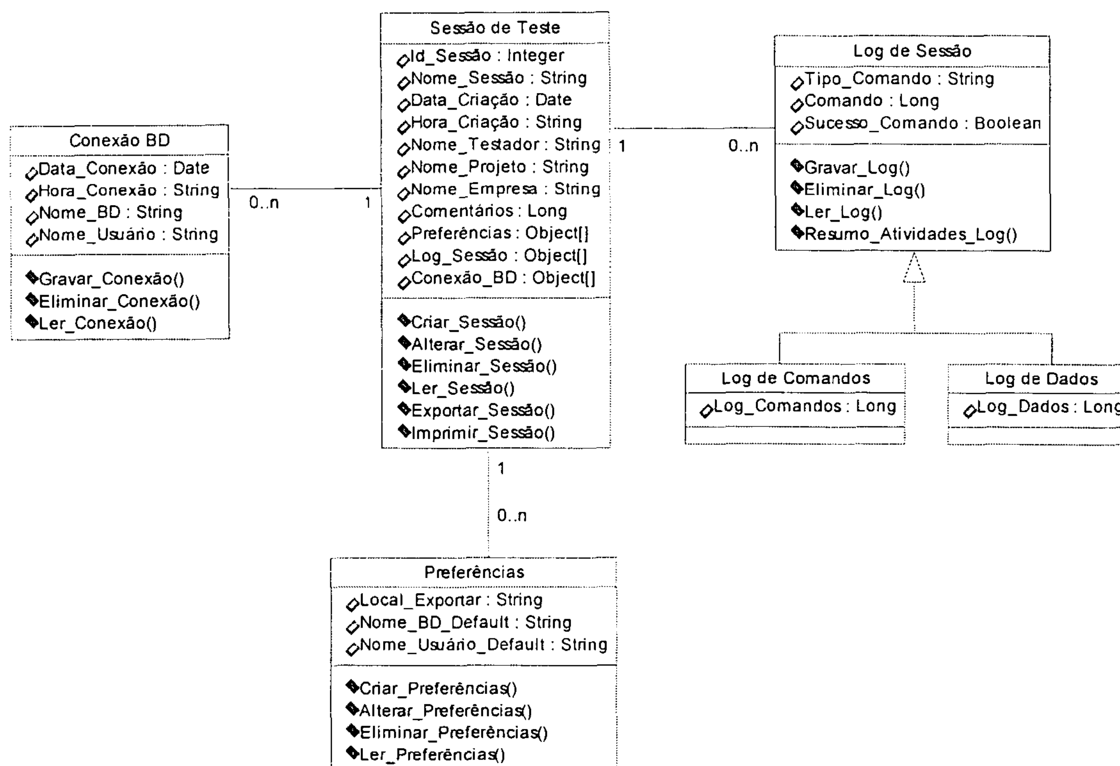


FIGURA 6 – DIAGRAMA DE CLASSES DA DBValTool

O diagrama entidade-relacionamento (DER) da Ferramenta, representando o modelo conceitual, é apresentado na Figura 7. As tabelas geradas segundo esse diagrama são usadas pela Ferramenta para armazenar todas as informações das sessões de teste.

A interface gráfica e amigável da Ferramenta integra todas as suas funcionalidades em um único ambiente, com ícones, formulários e relatórios, para tornar o teste mais fácil e intuitivo.

Apresenta-se, a seguir, uma descrição de uso da DBValTool por um testador, a fim de exemplificar a dinâmica de utilização da Ferramenta:

1. Primeiramente, o testador inicia uma sessão de teste que possibilitará a execução dos casos de teste e conterá as informações relacionadas a esses testes (nome do testador, projeto em teste, nome da empresa, conexões com o banco de dados, histórico de comandos, histórico de dados, etc). Se desejar, o testador poderá criar uma ou várias sessões durante os testes.

2. Em seguida, com o plano de teste nas mãos, o testador executa os casos de teste sobre a base de dados. A(s) operação(ões) SQL necessária(s) para a execução de um caso de teste pode(m) ser digitada(s) diretamente na janela Entrada SQL ou construída(s) por meio do Assistente, disponível na Ferramenta. Operações SQL mais elaboradas como, por exemplo, de junção, união de conjuntos, etc, estão disponíveis apenas via Entrada SQL.

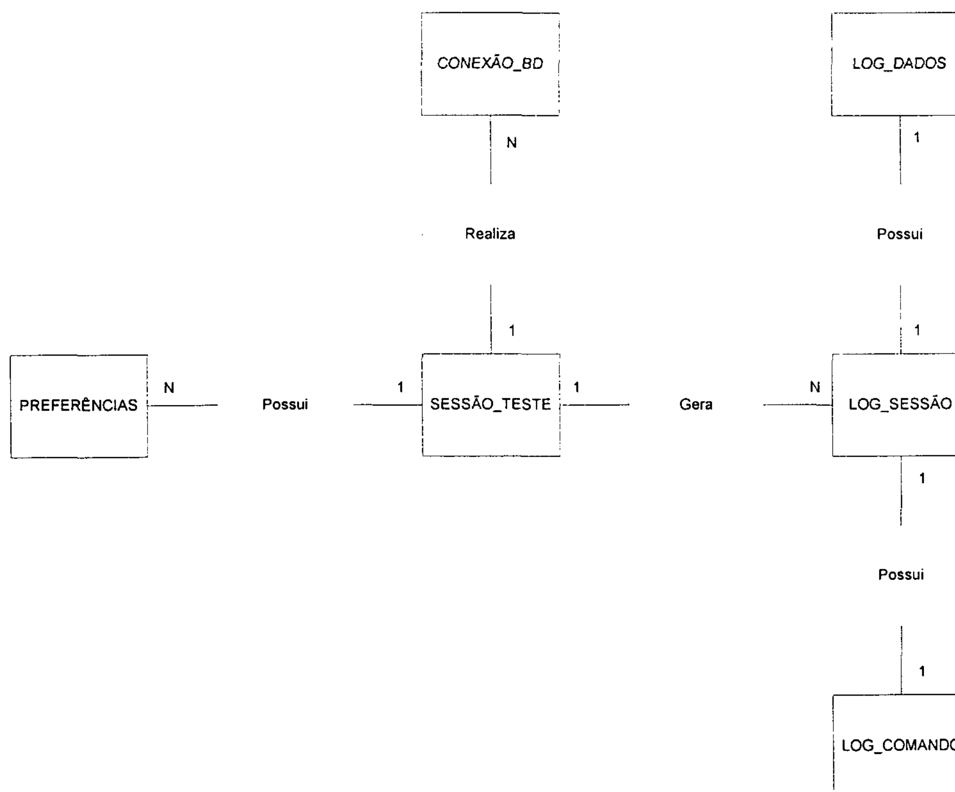


FIGURA 7 – DIAGRAMA ENTIDADE-RELACIONAMENTO (DER) DA DBValTool

3. Via Assistente, o testador deve selecionar o *schema*-tabela a ser testado. Na seqüência, a Ferramenta apresenta um formulário para a entrada dos dados de teste e, após a confirmação do testador, gera e executa uma operação SQL, sem erros de sintaxe.
4. As operações executadas e os seus resultados (código de retorno do banco de dados, mensagens de erros, *triggers* disparadas, *procedures* executadas, etc) são apresentados no histórico de comandos. No caso de execução sem erro, as modificações provocadas na base de dados são apresentadas no

histórico de dados.

5. Durante e após a execução dos casos de teste, o testador pode utilizar-se da Ferramenta, por meio do relatório de atividades e das consultas aos históricos gerados, para fazer avaliações dos resultados obtidos, especialmente em termos de possíveis erros.
6. Se falhas observáveis forem produzidas, o testador deve procurar o defeito causador na especificação do projeto da base de dados.
7. Durante e após a execução dos casos de teste, o testador pode executar o Verificador de *Warnings* de Projeto para que a Ferramenta apresente as possíveis advertências em relação à especificação do projeto da base de dados.

A Figura 8 apresenta a utilização da DBValTool durante o gerenciamento das sessões de teste.

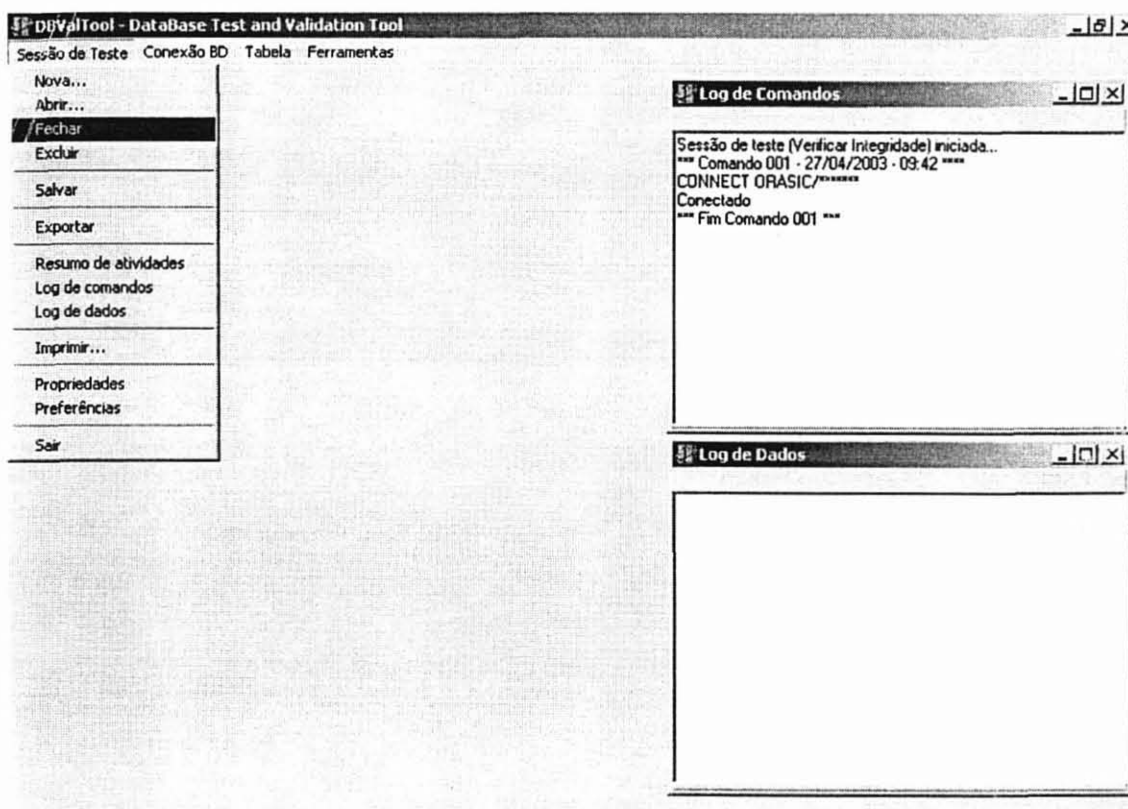


FIGURA 8 – UMA VISÃO DO GERENCIAMENTO DE SESSÕES DE TESTE NA DBValTool

A Figura 9 ilustra a utilização da DBValTool durante a execução de um caso de teste, via Assistente.

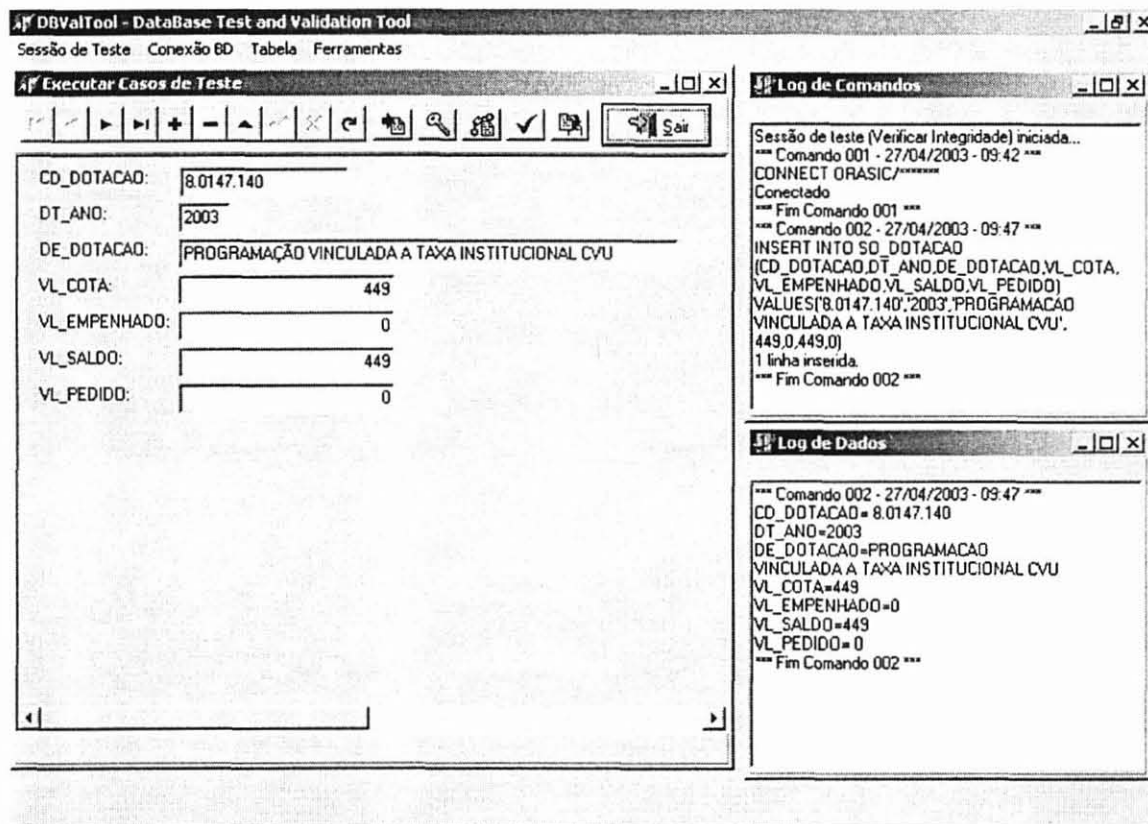


FIGURA 9 – UMA VISÃO DA EXECUÇÃO DE UM CASO DE TESTE NA DBValTool

4.2.4 *Warnings* de Projeto de Banco de Dados Propostos e Implementados

A experiência profissional adquirida ao longo dos anos como desenvolvedor de sistemas e projetista de banco de dados revela-nos que muitos defeitos são inseridos no projeto de banco de dados em decorrência de o projetista desconsiderar ou esquecer de verificar alguns aspectos básicos, mas importantes, na elaboração dos projetos.

A proposta, no sentido de diminuir ou eliminar esses defeitos, é a criação de uma lista de advertências aos projetistas, composta dos defeitos historicamente mais observados, para que seja sistematicamente utilizada na avaliação dos projetos de banco de dados. Durante a etapa de teste, essas advertências seriam avaliadas pelo

testador, uma vez que podem ser ou vir a ser um defeito causador de erros. Chamaremos essas advertências de *warnings* de projeto.

A idéia é que, a partir de um conjunto inicial, a lista de *warnings* pode ser ampliada ou reduzida, de acordo com as características e necessidades de cada organização. Não nos parece interessante reduzir a lista, mas sempre aumentá-la. A verificação desses *warnings* de forma automatizada, suportada por uma ferramenta, pode tornar esse processo mais simples e rápido.

Elaborou-se uma lista inicial de sete advertências, que julgamos ser causa freqüente de erros em muitos projetos de bases de dados. Essas advertências, implementadas pela DBValTool, são descritas a seguir.

W01: Atributo de relação com as restrições de integridade *not null* e *unique* sem a declaração de chave primária na relação

Chave primária é o atributo, ou grupo de atributos, que permite a identificação única de uma tupla em uma relação. Assim, toda chave primária deve ser não nula e única na relação. A existência de atributo com essas duas restrições, em uma relação sem a declaração de chave primária, é um indício de que se trata de atributo de chave primária. Nesse caso, a declaração de chave primária é desejável.

W02: Relação sem a declaração de chave primária

Relações são comumente apresentadas como tabelas nas quais cada linha representa dados sobre uma entidade particular e cada coluna representa um aspecto particular desses dados. Relação equivale à noção de conjunto, ou seja, um agrupamento de elementos sem repetição. Assim, torna-se desejável que toda relação possua uma chave primária declarada.

W03: Relações com atributos de mesmo nome e diferentes definições

De acordo com o dicionário de dados de um projeto de base de dados, um atributo deve ser definido uma única vez e suas características (nome, tipo de dado, tamanho, etc) devem ser as mesmas independentemente da quantidade de vezes e do

local onde é referenciado. Assim, torna-se indesejável a existência, em diferentes relações, de atributos de mesmo nome e diferentes definições.

W04: Relações com atributos de mesmas características sem a declaração de chave estrangeira

A existência de atributos de mesmas características (nome, tipo de dado, tamanho, etc), em diferentes relações, pode indicar a presença de uma chave estrangeira em uma relação, quando houver a declaração de chave primária para o mesmo atributo em outra relação. Nesse caso, quando se tratar do mesmo atributo, torna-se necessária a declaração de chave estrangeira, a fim de manter a integridade referencial.

W05: Declaração de chave estrangeira sem a indicação da ação (*restrict, delete, cascade, set null*) a ser adotada nos casos de exclusão ou atualização da chave primária referenciada

A atualização ou exclusão da chave primária de uma relação deve gerar algum tipo de ação (*restrict, delete, cascade, ser null*) na(s) chave(s) estrangeira(s) a ela referenciada(s). A indicação da ação é desejável por constituir-se em um dispositivo de manutenção da integridade referencial.

W06: Integridade referencial "circular"

Esse tipo de integridade referencial caracteriza-se por estabelecer um "círculo" de restrições de integridade referencial entre várias relações. Quando os atributos envolvidos nas declarações possuírem a restrição de integridade *not null*, esse tipo de integridade torna-se indesejável por causar uma anomalia na manutenção dessas relações, impedindo, por exemplo, a inclusão de novas tuplas. A Figura 10 apresenta um exemplo desse tipo de *warning*. Nesse exemplo, a tabela EMP, por meio do atributo **Deptno** (FK), possui um relacionamento com a tabela DEPT; por sua vez, a tabela DEPT, por meio do atributo **Mgr** (FK), possui um relacionamento com a tabela EMP.

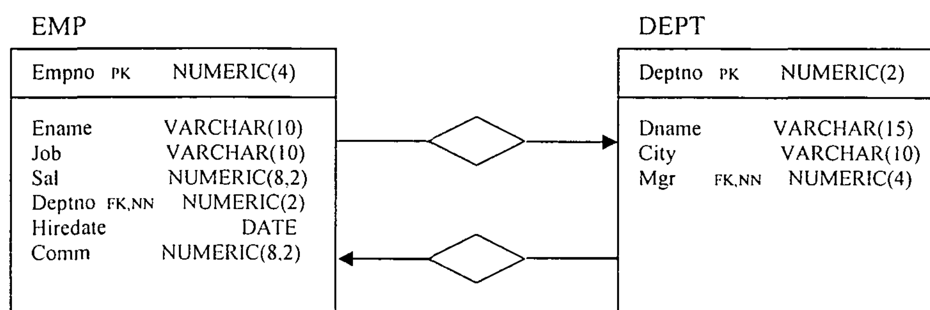


FIGURA 10 – EXEMPLO DE *WARNING* DO TIPO W06

W07: Integridade referencial "hierárquica" ou "auto-relacionamento"

Esse tipo de integridade referencial caracteriza-se por estabelecer uma restrição com a própria relação em que foi declarada e pode ser considerada uma variação do *warning* W06. A diferença principal em relação ao W06 é que envolve uma única relação e colunas de mesmas características e diferentes nomes. A Figura 11 apresenta um exemplo desse tipo de *warning*. Nesse exemplo, a tabela EMP possui um auto-relacionamento no qual o atributo **Mgr** é uma chave estrangeira (FK) da própria tabela EMP; neste caso, EMP passa a ser uma tabela hierárquica.

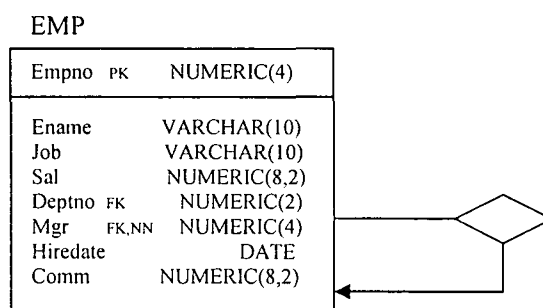


FIGURA 11 – EXEMPLO DE *WARNING* DO TIPO W07

4.3 CONSIDERAÇÕES FINAIS

Neste capítulo, são apresentadas as principais contribuições deste trabalho: o modelo de processo de validação de projeto e a Ferramenta DBValTool.

O modelo de processo de validação apresentado propõe um conjunto sistemático e planejado de atividades, com o objetivo principal de aumentar a qualidade e a confiabilidade do projeto.

A Ferramenta DBValTool proporciona, por meio das funcionalidades implementadas, um ambiente para teste de bases de dados relacionais.

No próximo capítulo, são apresentados os dois estudos de caso realizados para a validação deste trabalho.

5 VALIDAÇÃO DO MODELO PROPOSTO E DA FERRAMENTA DBValTool

Neste capítulo, são descritos os testes realizados para se verificar a eficiência do modelo e da ferramenta propostos neste trabalho. Primeiramente, apresenta-se a metodologia empregada nos testes e, em seguida, faz-se a descrição dos procedimentos realizados e dos resultados obtidos.

5.1 METODOLOGIA

Foram realizados dois estudos de caso (ECs), doravante chamados EC-I e EC-II. Para ambos os casos, foram selecionados dois testadores, de agora em diante denominados T-A e T-B, de experiências profissionais e de formação acadêmica semelhantes, para que eventuais diferenças na realização dos testes não pudessem ser atribuídas a diferenças na qualificação dos testadores. A finalidade dos estudos de caso é validar a proposta de modelo de processo de validação de projeto, bem como a Ferramenta DBValTool.

EC-I foi realizado com um projeto de pequeno porte, de aplicação comercial, desenvolvido por uma empresa do setor público. Trata-se de um sistema de controle financeiro dos pedidos de compra. Por sua vez, EC-II foi realizado com um sistema de controle patrimonial, ou seja, também é uma aplicação comercial, mas de médio porte, desenvolvido por uma empresa do setor privado. Apresenta-se, no Quadro 1, um resumo quantitativo das definições e declarações formais encontradas em cada um dos projetos selecionados.

Como se pode observar no Quadro 1, por meio das quantidades apresentadas, EC-II é um projeto mais complexo do que EC-I, uma vez que apresenta uma quantidade muito maior de relações, de atributos, de chaves primárias, de chaves estrangeiras e de restrições. EC-I não apresenta nenhuma ocorrência de restrição do tipo *unique*, nem de *triggers*, *procedures* e *functions*.

Em EC-I, T-A foi orientado a realizar os testes de maneira habitual, uma vez que a empresa não adota nenhuma metodologia ou padrão para esse tipo de

experimento. Por outro lado, solicitou-se a T-B que adotasse o modelo proposto de validação de projeto, apoiado pela Ferramenta DBValTool.

QUADRO 1 – RESUMO QUANTITATIVO DAS DEFINIÇÕES E DECLARAÇÕES FORMAIS ENCONTRADAS NOS PROJETOS

Definições Encontradas nos Projetos	EC-I	EC-II
Relações	9	147
Atributos	77	1336
Chaves primárias	9	128
Chaves estrangeiras	2	213
Restrições do tipo *not null*	53	841
Restrições do tipo *unique*	0	4
Restrições do tipo *check*	26	701
Triggers	0	62
Procedures	0	1077
Functions	0	1063

Em EC-II, houve uma inversão na maneira de papéis dos testadores. Solicitou-se a T-A que adotasse o modelo proposto de validação de projeto, apoiado na Ferramenta DBValTool. Por sua vez, T-B foi orientado a realizar os experimentos de acordo com as normas e padrões da empresa.

Os ECs foram divididos em duas etapas. Na primeira etapa, os testadores realizaram três atividades, ou seja, elaboraram os casos de teste, executaram os casos de teste e, por fim, avaliaram os resultados obtidos. Na segunda etapa, que foi realizada apenas pelos testadores que não utilizaram o modelo proposto neste trabalho nem a Ferramenta DBValTool (T-A, em EC-I; T-B, em EC-II), foram executadas novamente duas atividades: execução dos casos de teste e avaliação dos resultados obtidos, mas dessa vez com o apoio da Ferramenta DBValTool.

5.2 PROCEDIMENTOS

O critério utilizado para a elaboração dos casos de testes foi o de exercitar, pelo menos uma vez, todos os tipos de operações de definição (*insert*, *update* e *delete*) e todas as restrições especificadas (com a ativação e a não-ativação das restrições), em

todas as relações e atributos. Em consenso, foi acordado que, em cada EC, a quantidade de casos de teste a serem utilizados nos experimentos seria a mesma, para que essa variável não influenciasse no resultado dos testes e simplificasse a sua interpretação. O testador que utilizou a Ferramenta desfrutou da possibilidade de realização de consultas ao dicionário do banco de dados, dispensando, muitas vezes, a necessidade de consulta à documentação do sistema. O outro testador utilizou exclusivamente a documentação do sistema.

Na execução dos casos de teste, o testador que não utilizou o modelo e a ferramenta propostos neste trabalho digitou todas as operações por meio do SQL*Plus® (ferramenta que acompanha o gerenciador Oracle), direcionando a saída (resultado da execução) para um arquivo texto (.TXT), no sistema operacional, para análise futura. O testador que utilizou o modelo e a ferramenta propostos neste trabalho, por sua vez, decidiu criar apenas uma sessão de teste para todo o EC e utilizou o assistente da Ferramenta para a entrada das operações. Os resultados obtidos eram gravados nos históricos de comandos e nos históricos de dados, juntamente com mensagens de erros, definições das *triggers*, *procedures* e *functions* executadas, etc, sem necessidade de busca futura dessas informações. Uma observação que deve ser feita é que em EC-I, T-B digitou algumas operações pela via Entrada SQL, mas rapidamente retornou para o Assistente devido à produtividade e às facilidades proporcionadas por ele.

Nos dois ECs, a especificação do projeto foi a base para a avaliação dos resultados pelos testadores. Em cada EC, o testador que não utilizou o modelo e a ferramenta propostos avaliou as informações geradas a partir do conteúdo do arquivo texto gerado na execução dos casos de teste e, quando necessário, utilizou o SQL-Plus para obter mais informações sobre as definições de, por exemplo, relações, *triggers*, etc. A maior dificuldade encontrada por esse testador, nos dois ECs, foi a de conferir quando as operações eram realizadas com sucesso, as tuplas envolvidas e os seus conteúdos. O testador que utilizou o modelo e a ferramenta propostos neste trabalho, por sua vez, utilizou exclusivamente a Ferramenta DBValTool para avaliação dos

resultados, consultando informações contidas nos seus históricos e no dicionário do banco de dados.

5.3 RESULTADOS

Um resumo quantitativo dos resultados encontrados em cada EC é apresentado no Quadro 2 a seguir.

QUADRO 2 – RESUMO QUANTITATIVO DOS RESULTADOS OBTIDOS

Atividades	EC-I			EC-II		
	Etapa 1		Etapa 2	Etapa 1		Etapa 2
	T-A	T-B	T-A	T-B	T-A	T-B
Elaboração dos casos de teste						
• quantidade de casos de teste elaborados	27	27	–	441	441	–
• tempo gasto na elaboração dos casos de teste	25min	18min	–	12h15min	09h45min	–
Execução dos casos de teste						
• quantidade de casos de teste executados	27	27	27	441	441	441
• tempo gasto na execução dos casos de teste	55min	36min	42min	17h20min	09h35min	08h55min
Avaliação dos resultados						
• tempo gasto na avaliação dos resultados	35min	25min	28min	10h55min	06h05min	05h45min
• quantidade de defeitos detectados	1	2	2	3	9	9
Verificação de *warnings* de projeto						
• quantidade de *warnings* relacionados	–	7	7	–	31	31
• quantidade de *warnings* identificadores de defeitos	–	1	2	–	4	4

Como se pode observar no Quadro 2, a utilização do modelo e da ferramenta propostos neste trabalho proporcionou, nos ECs, uma redução média de 34% no tempo gasto com os testes e uma detecção de uma maior quantidade de defeitos, comparando-se com o processo tradicional.

Em EC-I, o tempo gasto pelo testador que utilizou a Ferramenta foi em média 31% menor do que o tempo gasto pelo testador que não a utilizou. Na elaboração dos casos de teste, o tempo foi 28% menor. Na execução desses casos, o tempo gasto foi em média 29,1% menor, e, na avaliação dos resultados, 24,3% menor em média.

Em EC-II, o tempo gasto pelo testador que empregou a Ferramenta foi em média 37% menor do que o tempo gasto pelo testador que não a utilizou. Na elaboração dos casos de teste, o tempo foi 20,4% menor. Na execução dos testes, o

tempo gasto foi em média 46,6% menor, e, na avaliação dos resultados, o tempo foi em média 45,8% menor.

A maior redução de tempo (48,6%) foi encontrada na atividade de execução dos casos de teste, Etapa 2, no EC-II.

No que diz respeito aos defeitos (a lista de *warnings* encontra-se na seção 4.2.4), pode-se afirmar que o fato de cada testador ter elaborado os seus casos de teste não influenciou significativamente na quantidade final de defeitos encontrados. O Quadro 3 apresenta os tipos de defeitos e *warnings* encontrados em cada EC, com o apoio da DBValTool.

QUADRO 3 – TIPOS DE DEFEITOS DETECTADOS COM O APOIO DA DBValTool

Defeitos Detectados (com o apoio da DBValTool)	**EC-I**	**EC-II**
Atributo sem a restrição *not null*	1	1
Atributo de tipo *date* sem restrição de domínio	1	3
Atributo de tipo *integer* sem restrição de domínio	-	2
Chave primária não definida	-	1
Chave estrangeira não definida	-	1
Execução indevida de *trigger*	-	1
Warning W02	-	1
Warning W04	-	1
Warning W05	2	2

Em EC-I, na avaliação dos resultados, com o auxílio da Ferramenta, T-A e T-B identificaram os mesmos defeitos (2), um a mais que T-A sem o uso da Ferramenta. Os 2 defeitos encontrados com o auxílio da Ferramenta correspondem a 1 ausência de restrição *not null* em um atributo e 1 ausência de restrição de domínio para um atributo do tipo *date*, sendo que este não foi encontrado sem o uso da Ferramenta. Ainda no EC-I, os 7 *warnings* listados pela Ferramenta correspondem a 2 do tipo W03, 3 do tipo W04 e 2 do tipo W05. Os dois do tipo W05 correspondiam a defeitos. T-A conseguiu identificar apenas um deles.

Em EC-II, na avaliação dos resultados, com o auxílio da Ferramenta, T-A e T-B identificaram os mesmos defeitos (9), 6 a mais que T-B sem o uso da Ferramenta. Os 9 defeitos encontrados, com o auxílio da Ferramenta, correspondem a 3 ausências

de restrição de domínio para um atributo do tipo *date*, 2 ausências de restrição de domínio de atributos do tipo *integer*, 1 ausência de restrição do tipo *not null*, 1 falta de definição de chave primária, 1 falta de definição de chave estrangeira e 1 execução indevida de um *trigger*. Ainda em EC-II, os 31 *warnings* listados pela Ferramenta correspondem a 19 do tipo W02, 8 do tipo W03, 1 do tipo W04 e 2 do tipo W05. Desses *warnings*, 1 do tipo W02, 1 do tipo W04 e 2 do tipo W05 correspondiam a defeitos.

Uma observação importante a ser feita é que os resultados obtidos nos estudos de caso são fortemente influenciados pelos projetos selecionados. Assim, a generalização e a utilização desses resultados devem ser feitas de forma cuidadosa.

5.4 CONSIDERAÇÕES FINAIS

A DBValTool oferece um ambiente integrado com o banco de dados para o teste de bases de dados relacionais. Os resultados obtidos nos estudos de caso demonstraram que o uso da Ferramenta DBValTool, durante todo o processo de validação do projeto, proporcionou uma redução média de, aproximadamente, 34% do tempo total destinado às atividades de teste e também auxiliou na detecção de uma quantidade maior de defeitos nos projetos.

Finalmente, no próximo capítulo, são apresentadas as conclusões e as contribuições deste trabalho.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo, são apresentadas as conclusões e contribuições deste trabalho e, finalmente, são propostos e discutidos os trabalhos futuros de pesquisas.

6.1 CONCLUSÕES

Diante do importante papel que os bancos de dados desempenham nas modernas organizações, verifica-se a necessidade do desenvolvimento e do aprimoramento de métodos, técnicas e ferramentas para apoiar o teste de bases de dados, principalmente antes de serem povoadas. O enfoque principal deste trabalho é o teste e a validação do projeto de banco de dados relacional, com o objetivo de assegurar que o projeto seja implementado da forma como a sua especificação determina.

É proposto um modelo de processo de validação de projeto que permite avaliar o quanto estão corretas e completas as definições criadas no banco de dados, visando detectar possíveis erros na criação dos esquemas relacionais, antes de passar para etapas seguintes do processo de desenvolvimento. Assim, sugere-se que um projeto de aplicação de banco de dados seja iniciado com uma avaliação da base de dados, a fim de garantir que essa base esteja validada, para, só então, partir para o desenvolvimento da aplicação.

É proposta e implementada uma ferramenta – a DBValTool – para apoiar o teste de bases de dados, visando proporcionar um aumento da qualidade e uma redução no tempo e no custo dessa atividade. A Ferramenta pode apoiar, também, o modelo de processo de validação de projeto proposto, com funcionalidades que podem ser utilizadas nas várias etapas do processo de validação, buscando proporcionar um aumento da qualidade e da confiabilidade do projeto. Dentre as suas funcionalidades, destacam-se, a de apoiar o teste de bases de dados, por meio da execução de casos de teste e da análise de seus resultados, e a de analisar informações extraídas do esquema

da base de dados, emitindo advertências que podem ser ou que podem se tornar um defeito causador de erros.

A validação do modelo proposto e da Ferramenta DBValTool foi realizada por meio de dois estudos de caso, apresentados neste trabalho. Como se esperava inicialmente, a utilização do modelo, apoiado pela Ferramenta, apresentou contribuições ao processo de teste e de validação do projeto.

Os resultados obtidos, nos estudos de caso realizados, demonstraram contribuições para a redução do tempo total destinado às atividades de teste e para a detecção de defeitos na definição dos esquemas das bases de dados e na especificação do projeto dessas bases. Esses resultados proporcionaram um aumento da qualidade e da confiabilidade das bases de dados e do projeto. Outro resultado obtido foi a conscientização da necessidade desse tipo de teste, muitas vezes não realizado nas organizações.

Não se pode deixar de considerar que os resultados obtidos nos estudos de caso são fortemente influenciados pelos projetos selecionados. Assim, a generalização e a utilização desses resultados devem ser feitas de forma cuidadosa.

6.2 CONTRIBUIÇÕES DESTE TRABALHO

As contribuições deste trabalho estão relacionadas à proposição de um modelo de processo de validação de projeto, bem como à proposição e ao desenvolvimento de uma ferramenta para apoiar o teste de bases de dados, além da realização de dois estudos de casos para a validação do modelo proposto e da Ferramenta DBValTool. As principais contribuições são apresentadas, de forma objetiva, a seguir:

- sistematização do processo de validação de projeto de banco de dados;
- proposição de mecanismos para assegurar que o projeto de banco de dados seja implementado da forma como a sua especificação determina;
- desenvolvimento de uma ferramenta para apoiar o teste e a validação de projeto de banco de dados;
- demonstração prática do suporte automatizado ao processo de validação de

projeto de banco de dados;

- melhoria da qualidade e da confiabilidade do projeto de banco de dados;
- melhoria da qualidade da atividade de teste de bases de dados;
- redução no tempo e no custo da atividade de teste de bases de dados;
- conscientização e consolidação da necessidade e a da importância do teste de bases de dados e da validação de projeto de banco de dados, muitas vezes não realizados nas organizações;
- aprimoramento do processo de desenvolvimento de sistemas de banco de dados, como forma de suprir ou minimizar as dificuldades encontradas pelas organizações – principalmente as pequenas –, em decorrência das limitações de recursos que impossibilitam, por exemplo, a aquisição dos melhores, e normalmente mais caros, bancos de dados.

6.3 TRABALHOS FUTUROS

A expectativa é de que, à medida que a Ferramenta DBValTool seja apresentada a pessoas que trabalham com teste de base de dados e validação de projeto, surjam novas sugestões de funcionalidades. Assim, é natural que ela esteja continuamente se expandindo e se aperfeiçoando, tentando suprir as necessidades dos testadores. Algumas funcionalidades já sugeridas e outras vislumbradas, para implementação futura, são:

- disponibilização para uso com os bancos de dados DB2, SQLServer, MySQL e PostgreSQL;
- implementação de alguns critérios de teste;
- aprimoramento do assistente de execução dos casos de teste para que aceite operações SQL mais elaboradas.

Em relação aos *warnings* de projetos de banco de dados apresentados neste trabalho propomos trabalhos futuros no sentido de:

- ampliar a lista de *warnings* a serem verificados pela Ferramenta, abrangendo tanto os aspectos estáticos quanto os dinâmicos dos bancos de

dados;

- criar uma base de dados de *warnings*, na qual estes seriam definidos por uma linguagem de definição de *warnings*, e disponibilizá-la para o uso por outras ferramentas.

Outras sugestões de trabalhos futuros, também relacionadas a este trabalho, são:

- estudos no sentido de estender as contribuições deste trabalho aos bancos de dados orientados a objeto e aos bancos de dados objeto-relacionais;
- criação de uma "máquina" de validação de projetos de banco de dados. A idéia é que os projetistas e testadores, local ou remotamente, submetam seus projetos para uma avaliação do quanto estão corretas e completas as definições criadas no banco de dados, tudo isso de acordo com padrões previamente estabelecidos;
- criação de *guidelines* para auxiliar na V&V de um projeto lógico e físico de banco de dados relacional aplicado durante o processo de validação do projeto de banco de dados, durante a fase de desenvolvimento de um produto de *software* com uso de banco de dados.

REFERÊNCIAS

ARANHA et al. **RDBTool: Uma Ferramenta de Apoio ao Teste de Bases de Dados Relacionais**. Curitiba: XI CITS, 2000.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. **The Unified Modeling Language User Guide.** New York: Addison-Wesley, 1999.

BUENO, P. M. S. et al. **Um Estudo de Critérios de Teste de Software Baseados em Fluxo de Dados**. Campinas: UNICAMP-FEE-DCA, Trabalho da Disciplina Tópicos de Engenharia da Computação, 1995.

CHAYS et al. **A Framework for Testing Database Applications**. IEEE Transactions on Software Engineering, Agosto, 2000.

CHEN, P. **Modelagem de Dados, a Abordagem Entidade-Relacionamento para o Projeto Lógico**. São Paulo: Makron Books, 1992.

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

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 3. ed. New York: Addison-Wesley, 2000.

JAHNE, A.; URBAN, S. D.; DIETRICH, S. W. **Peard: A Prototype Environment for Active Rule Debugging.** Journal of Intelligent Information Systems, Special Issue on Active Database Systems , Volume 7, Issue 2, October 1996.

LEWIS, R. O. **Verification e Validation: a life cycle engineering process for quality software**. New York: Wiley-Interscience, 1992.

MALDONADO, J.C. **Critérios Potenciais Usos: uma Contribuição ao Teste Estrutural de Software**. Campinas: UNICAMP-FEE-DCA, Tese de Doutorado, 1991.

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

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

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

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1998.

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

DOCUMENTOS CONSULTADOS

ANDRADE, M. J. B. **Manutenção de Restrições de Integridade em Banco de Dados Orientados a Objetos.** Campinas: UNICAMP, Instituto de Matemática, Estatística e Ciência da Computação, Dissertação de Mestrado, 1992.

CHAIM, M. L. **POKE-TOOL – Uma Ferramenta para Suporte ao Teste Estrutural de Programa Baseado em Análise de Fluxo de Dados**. Campinas: UNICAMP-FEE-DCA, Dissertação de Mestrado, 1991.

GARVIN, D. A. **Gerenciando a Qualidade: Visão estratégica e Competitiva**. Rio de Janeiro: Qualitymark, 1992.

MACDERMID, J. A.; ROOK, P. **Software Development Process Models, Software Engineering Reference Book**. Ed. J. A. McDermid, Butterworth-Heineman Scientific, 1991.

MEDEIROS, C.; PFEFFER, P. **Transformação de um Banco de Dados Orientado a Objetos em um Banco de Dados Ativo**. VI Simpósio de Banco de Dados, Março, 1991.

UNIVERSIDADE FEDERAL DO PARANÁ. Biblioteca Central. **Normas para Apresentação de Documentos Científicos**. Curitiba, Editora UFPR, 1992, Volumes 2, 6, 7 e 8.

ANEXO 1 - UML

Este anexo apresenta uma revisão da *Unified Modeling Language* (UML). O propósito não é um estudo detalhado sobre UML, mas uma breve exposição das suas principais características e notações, utilizadas para apresentação da ferramenta desenvolvida. Um estudo completo sobre UML pode ser obtido em *The Unified Modeling Language User Guide* (BOOCH; JACOBSON; RUMBAUGH, 1999). O texto a seguir baseia-se nessa referência bibliográfica.

A UML é uma linguagem usada para especificação, construção, visualização e documentação de sistemas de *software*, assim como para a modelagem de negócios e outros sistemas não baseados em *software*. A UML é uma evolução dos conceitos de Booch, OMT (*Object Modeling Technique*) de Rumbaugh e OOSE (*Object-Oriented Software Engineering*) de Jacobson e também de outros métodos orientados ou não a objetos. A notação UML é uma combinação da sintaxe gráfica de vários métodos, com alguns símbolos removidos e com alguns símbolos adicionados. É uma linguagem de modelagem única, comum e ampla, utilizável por usuários destes e de outros métodos.

No desenvolvimento de sistemas de *software*, a UML apresenta cinco fases distintas: análise de requisitos, análise, *design* (projeto), programação e testes. Essas cinco fases não devem ser executadas na ordem descrita anteriormente, mas concomitantemente, para que problemas detectados numa certa fase modifiquem e melhorem as fases desenvolvidas anteriormente, para que o resultado global gere um produto de alta qualidade e desempenho.

A partes que compõem a UML são:

- visões: mostram diferentes aspectos do sistema que está sendo modelado. A visão não é um gráfico, mas uma abstração consistindo em uma série de diagramas. Definido um número de visões, cada uma mostrará aspectos particulares do sistema, dando enfoque a ângulos e níveis de abstração diferentes e uma figura completa do sistema poderá ser construída. As visões também podem servir de ligação entre a linguagem de modelagem e o método/processo de desenvolvimento escolhido;

- modelos de elementos: os conceitos usados nos diagramas são modelos de elementos que representam definições comuns da orientação a objetos como as classes, objetos, mensagem, relacionamentos entre classes incluindo associações, dependências e heranças;
- mecanismos gerais: provêem comentários suplementares, informações, ou semântica sobre os elementos que compõem os modelos; eles provêem também mecanismos de extensão para adaptar ou estender a UML para um método/processo, organização ou usuário específico;
- diagramas: os diagramas são os gráficos que descrevem o conteúdo em uma visão. A UML possui nove tipos de diagramas que são usados em combinação para prover todas as visões do sistema: diagrama de *use cases* (casos de uso), de classes, de objeto, de seqüência, de colaboração, de estados, de atividades, de componentes e o de implantação. Esses vários diagramas que formam a notação da UML incorporam diferentes elementos propostos por vários autores.

Tendo em mente as cinco fases do desenvolvimento de *software*, as fases de análise de requisitos, análise e *design* utilizam-se, em seu desenvolvimento, dos cinco tipos de visões, dos nove tipos de diagramas e de vários modelos de elementos que serão utilizados na criação dos diagramas e mecanismos gerais que, em conjunto, especificam e exemplificam a definição do sistema.

Para modelar um sistema, a escolha das técnicas que serão utilizadas dependerá de como um problema é considerado e como uma solução é encontrada. A UML é intencionalmente independente de processo, não definindo um processo padrão para sua utilização.

Para usar a UML com sucesso é necessário adotar algum tipo de método de desenvolvimento, especialmente em sistema de grande porte onde a organização de tarefas é essencial. A utilização de um processo de desenvolvimento torna mais eficiente calcular o progresso do projeto, controlar e melhorar o trabalho.

Como já foi esclarecido no início deste anexo, não é objetivo deste trabalho apresentar um estudo detalhado sobre UML e seus diagramas. Limitamo-nos apenas a

descrever alguns conceitos básicos sobre essa linguagem e apresentar os diagramas de *use cases* e de classes, que serão utilizados para descrever a ferramenta desenvolvida.

Diagrama de *Use Cases*

Um diagrama de *use cases* utiliza uma coleção de casos de usos e atores para especificar ou caracterizar a funcionalidade e o comportamento de um sistema de aplicação. Os usuários ou quaisquer outros sistemas que possam interagir com o sistema que está sendo modelado são chamados de atores. Os atores representam os usuários do sistema, eles ajudam a delimitá-lo e fornecem uma visão clara do que será realizado.

Os diagramas de *use cases* contêm elementos representando os atores, relacionamentos de associação, relacionamentos de generalização e *use cases*. Pode ser criado um diagrama de *use cases* de alto nível para visualizar o contexo e limites do sistema. É possível também criar um ou mais diagramas de *use cases* para descrever um sistema em partes.

Um ator é um estereótipo de uma classe e pode ser mostrado com uma representação especial de agente externo em um diagrama de *use cases*. Um ator modela um tipo de objeto que interage diretamente com o sistema mesmo estando fora do limite do sistema. Um ator inicia um caso de uso.

Um caso de uso é uma seqüência de transações realizadas pelo sistema em resposta ao disparo de um evento, do ator para o sistema. Um caso de uso, quando realizado completamente, fornece um valor avaliável para o ator. Os casos de uso explicam a interação do usuário com o sistema e representam os passos necessários para alcançar os resultados desejáveis para essa interação. Casos de uso não são independentes. Eles podem se sobrepor, ocorrer simultaneamente, ou influenciar uns aos outros. Casos de uso ocorrem sob condições específicas. O nível de abstração dos casos de uso e seu tamanho são conceitos arbitrários. Os casos de uso podem se relacionar com atores e com outros casos de uso.

O relacionamento de associação representa uma conexão semântica entre caso de uso e ator, é unidirecional, é mais comum que a generalização e o seu nome é usado para identificar o propósito do relacionamento.

Dependências entre casos de uso podem ser definidas usando os relacionamentos *include* e *extend*. O tipo <<include>> é usado para descrever o comportamento comum entre dois ou mais casos de usos. É um dos mecanismos utilizados para identificar comportamentos reutilizáveis pelas regras de negócio. O tipo <<extend>> é usado para expressar comportamento opcional por um caso de uso.

A generalização é um tipo de relacionamento entre casos de uso em que um caso de uso pode compartilhar o comportamento definido em um ou mais casos de uso. É similar ao relacionamento <<extend>> porque ele também é usado para indicar variações.

A Figura 12 apresenta a notação UML para um diagrama de *use cases*.

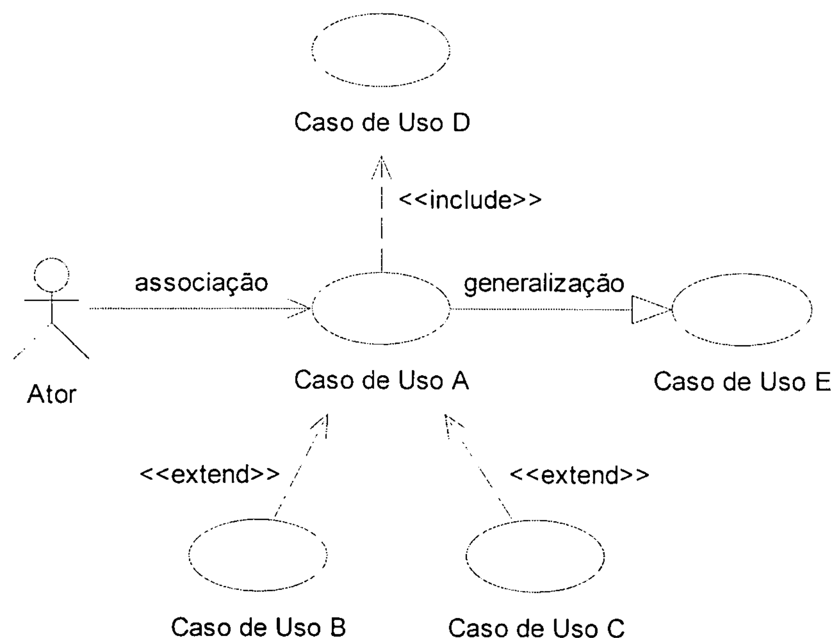


FIGURA 12 – NOTAÇÃO UML PARA UM DIAGRAMA DE *USE CASES*

Diagrama de Classes

Um diagrama de classes é uma representação gráfica utilizada para descrições genéricas do sistema. Esse diagrama pode conter tipo, relacionamentos e instâncias de classes. Uma classe captura a estrutura e o comportamento comum de um conjunto de objetos e é uma abstração de elementos do mundo real. Quando esses elementos existem no mundo real, são instâncias de classe e são referidos como objetos. Para cada classe que tem comportamento temporal significativo pode ser criado um diagrama de estado para descrever esse comportamento. A Figura 13 apresenta a notação UML para a representação das classes.

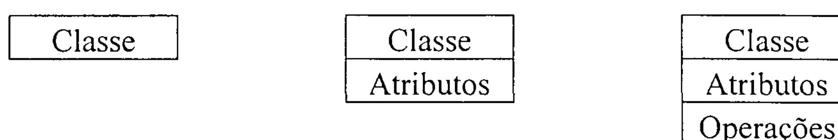


FIGURA 13 – NOTAÇÃO UML PARA AS CLASSES

A cardinalidade especifica o número de instâncias de uma classe em relação a outra, em um relacionamento, e é representada por anotações próximas nas linhas que conectam o relacionamento às classes. O Quadro 4 resume as possíveis variações de cardinalidade.

QUADRO 4 – NOTAÇÕES DE CARDINALIDADE

Notação	Significado
1	Exatamente uma instância.
0..1	Zero ou uma instância (opcional).
*	Zero ou mais instâncias.
1..*	Uma ou mais instâncias.
m..n	Número de instâncias numericamente especificado.

Os relacionamentos que podem existir em um diagrama de classes são: generalização, dependência, associação e agregação. Um relacionamento de

generalização entre classes mostra que a subclasse compartilha a estrutura ou comportamento definido em uma ou mais superclasses. Um relacionamento de dependência entre duas classes mostra que uma classe depende da outra para sua existência. Um relacionamento de associação representa uma conexão semântica entre duas classes, é bidirecional e é o relacionamento mais utilizado. Um relacionamento de agregação é uma forma especial de associação usada para mostrar que um tipo de objeto é composto, no mínimo, por uma parte de outro.

A agregação pode ser implementada na UML como agregação por referência ou agregação por valor. Na agregação por referência o "objeto todo" mantém um ponteiro ou uma referência para suas "partes". A agregação por valor indica que o tempo de vida das "partes" é dependente do tempo de vida do "todo".

A notação dos vários relacionamentos é apresentada, por meio de exemplos, na Figura 14.

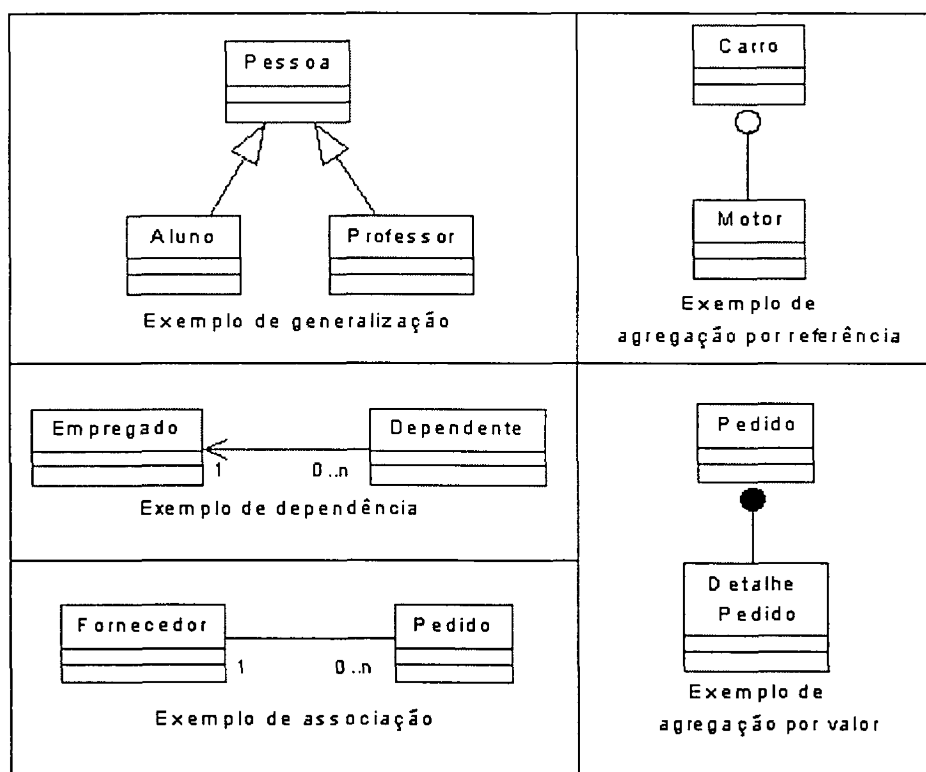


FIGURA 14 – EXEMPLOS DOS TIPOS DE RELACIONAMENTOS NOS DIAGRAMAS DE CLASSES, UTILIZANDO NOTAÇÃO UML.

ANEXO 2 - TESTE DE PROGRAMAS

Tradicionalmente, o teste está associado ao processo de executar o programa com o objetivo de encontrar defeitos (MYERS, 1979), seja em sua estrutura, seja em sua funcionalidade.

O teste de *software* tem como objetivo essencial revelar o maior número possível de defeitos existentes em um programa. Basicamente, isso pode ser realizado do seguinte modo: ao executar um programa deve-se fornecer dados de entrada e comparar a saída com o resultado esperado, obtido na especificação do *software*.

O sucesso dessa atividade está relacionado com a qualidade do conjunto de casos de teste. Nessa perspectiva, duas questões importantes são: "como selecionar casos de teste?" e "como garantir que o produto foi suficientemente testado?". Considerando que o teste exaustivo – em que o *software* é exercitado com todos os valores possíveis do domínio de entrada – é impraticável, critérios de teste são utilizados, os quais permitem selecionar um subconjunto do domínio de entrada preservando a probabilidade de revelar os erros existentes no produto. Esses critérios sistematizam a atividade de teste e podem também constituir uma medida de cobertura dessa atividade.

O texto a seguir, baseado em BUENO et al. (1995) e SPOTO (2000), apresenta alguns conceitos básicos em teste de programa e critérios de teste.

Conceitos Básicos

O processo de teste de programa, também conhecido como teste de *software*, consiste basicamente em introduzir dados de entrada necessários à execução do programa e, após sua execução, comparar os resultados obtidos com os resultados esperados.

O objetivo essencial do teste é o de revelar os defeitos existentes no programa. Naturalmente, o teste é uma forma de avaliar-se a qualidade do *software*. Nesse sentido, se nenhum defeito é encontrado, então a qualidade do *software* é assumida como sendo boa. Portanto, um propósito mais geral do teste consiste em aumentar a

confiança de que o *software* trabalha corretamente. Este intuito só é alcançável quando se utiliza uma metodologia de teste coerente; caso contrário, não há meios de saber se a ausência de falhas é mérito do *software* ou benemérito do teste.

Em sintonia com as idéias anteriores, pode-se dizer que um teste tem sucesso quando detecta defeitos no programa. Isso porque testes reveladores de defeitos dão ao testador muito mais informação do que os não reveladores. Como não é possível provar a corretude do programa através do teste, uma abordagem mais eficaz consiste em fazer o oposto: revelar que o programa está incorreto, através da utilização de dados de teste, que provoquem falhas no *software*. Esta idéia é geralmente relacionada com o Paradoxo do Teste. O aspecto paradoxal está exatamente em construir-se a confiança no *software*, tentando destruí-la, através da descoberta da existência de defeitos. Por fim, parece claro que não se pode concluir seguramente sobre a qualidade do *software* testado, sem algum conhecimento sobre a qualidade do teste efetuado. Isto motiva o estudo de como testar de maneira eficiente o *software*, e de como avaliar a qualidade do teste.

Cabe, neste momento, esclarecer alguns conceitos acima utilizados. Uma falha é um evento notável - ou perceptível - em que o sistema viola a sua especificação. Um defeito no programa é uma deficiência algorítmica que pode levar a uma falha do sistema. Um erro é um estado incorreto do conjunto de dados do programa, que pode ser manifestado, ocasionando assim uma falha. Na literatura, muitas vezes, utiliza-se a palavra erro neste sentido, mas também, em alguns casos, como sinônimo de defeito.

A situação ideal, para maximizar o número de defeitos revelados, é que se pudesse testar o programa com todos os dados do domínio de entrada. Isto seria o teste exaustivo. Mas este tipo de teste apresenta um grande problema que o torna inviável: caso o domínio de entrada seja infinito, o teste para aquele *software* nunca estaria completo; e, ainda, esse tipo de teste possui custos operacionais geralmente muito grandes, considerando que o número de combinações possíveis dos dados de entrada é potencialmente enorme.

Por tal motivo, casos de teste devem ser planejados, de tal modo a encontrar o máximo de defeitos com o menor tempo e esforço. Um caso de teste consiste na

especificação de dados de teste e sua respectiva saída esperada quando da execução do programa. Um dado de teste é um conjunto de valores de entrada do programa, selecionado do domínio de entrada, com o intuito de executar o programa em teste. A saída esperada é o conjunto de valores de saída que se espera que um *software* apresente, após sua execução, quando ele está correto.

O domínio de entrada é o conjunto total de valores de entrada para um determinado programa. Tais valores podem, de acordo com a especificação do programa, ser válidos ou inválidos. Os valores inválidos podem ser de invalidez por tipo de dado, ou invalidez pelo fato do valor estar fora da faixa especificada (a especificação do programa deve prever tratamento de exceção para esses valores inválidos). Para maior simplicidade, quando houver referência a domínio de entrada, entende-se o conjunto total de valores de entrada de válidos, acrescido do conjunto de valores inválidos por estarem fora da faixa especificada.

Após obter-se os resultados da execução de um programa, utiliza-se o oráculo, que é algum método, automatizado ou não, para determinar se a saída produzida pelo programa é correta ou não. Deve-se notar que o oráculo, em seu conceito mais geral, não fornece a saída correta, sendo apenas um método de avaliação da corretude da mesma.

Técnicas e metodologias para a geração de dados de teste utilizam-se de critérios que estabelecem o que vai ser testado no programa; induzindo a escolha de dados de teste que satisfaçam certas condições, que, a grosso modo, podem ser de natureza funcional – relacionadas à especificação de requisitos do programa – ou estrutural – relacionadas ao código do programa. Outro enfoque para a aplicação desses critérios, consiste em utilizá-los para verificar se essas condições foram satisfeitas (*a posteriori*), avaliando o nível de adequação dos dados utilizados.

Portanto, os dados de teste podem ser gerados segundo um critério, utilizando-se informações sobre o código, especificação do programa, informações históricas de erros comuns no processo de desenvolvimento, ou ainda, os dados podem ser gerados aleatoriamente. Em geral, quando um conjunto de dados de teste satisfaz todas as

condições estabelecidas por um dado critério de teste, diz-se que tal conjunto satisfez o critério, ou analogamente, que tal conjunto é adequado ao critério.

Uma estratégia de teste de *software*, pode consistir na aplicação sistematizada de diferentes critérios, de natureza distinta, e que revelam classes diversas de defeitos no programa.

No processo de teste de um sistema caracterizam-se três níveis de teste: o teste de unidade, o teste de integração e o teste de sistema. O teste de unidade tem como função detectar defeitos na menor unidade do *software*, isto é, o módulo. O teste de integração objetiva testar as relações e interfaces entre os módulos, sendo conduzido após o teste de unidade de todos os módulos envolvidos na integração. O teste de sistema, após o teste de integração, visa identificar erros de função e/ou características de desempenho (MALDONADO, 1991).

O programa pode ser avaliado segundo a visão funcional do *software* e a visão da estrutura da implementação. A técnica funcional de teste trata o programa como uma caixa preta (*black-box testing*), não interessando a implementação da especificação. Apenas o comportamento esperado é conhecido, a partir da especificação funcional do *software* (MALDONADO, 1991). A técnica estrutural de teste utiliza-se do conhecimento da estrutura interna do programa (*white-box testing*) e o objetivo é caracterizar um conjunto de componentes elementares de um programa que devem ser exercitados pelo conjunto de casos de teste (MALDONADO, 1991). É importante notar que as duas técnicas mencionadas não são alternativas, isto é, não se utiliza apenas uma ou outra técnica: elas são complementares. Geralmente, uma técnica proporciona a descoberta de classes de defeitos diferentes do que a outra proporciona. Por isso, para que o teste seja o mais completo possível, devem ser utilizadas duas técnicas.

Critérios de Teste

Para o teste possuir certa garantia de qualidade, e evitando o teste exaustivo, a alternativa é estabelecer certas condições que devam ser satisfeitas para que o teste

seja bem efetuado, minimizando o custo e maximizando a eficácia desse processo.

Entende-se por critério de teste um conjunto de condições que devam ser satisfeitas para que a atividade de teste seja realizada com sucesso. Independentemente do critério considerado, tem-se um conjunto de elementos requeridos pelo critério em que cada elemento requerido representa uma condição a ser satisfeita durante o teste.

Critérios existentes para o teste de *software* podem ser usados como critérios de parada, critérios de seleção de dados de teste e critérios de adequação de dados de teste.

Um critério de parada estabelece parâmetros que devem ser satisfeitos para que se encerre o processo de teste e, dessa forma, fornecendo uma condição de suficiência para o fim do processo de teste. Um critério de parada pode ser baseado em tempo, cobertura, número de erros restantes, entre outros.

Um critério de seleção de dados de teste estabelece parâmetros para a escolha dos dados de teste de um determinado domínio de entrada.

Um critério de adequação de dados de teste estabelece parâmetros para que se possa considerar um conjunto de dados de teste apropriado. É um meio de avaliar o quanto um conjunto de dados de teste utilizado satisfaz as condições impostas pelo critério.

Existe uma correspondência entre a seleção de dados de teste e a adequação dos mesmos a algum critério. Para um dado critério de adequação, existe um critério de seleção que busca satisfazê-lo. Nada impede, entretanto, que se utilize um determinado critério de teste para selecionar os dados, e um outro para medir a adequação dos mesmos.

Satisfazer um critério de teste significa exercitar todos os elementos requeridos pelo critério, para um dado programa, mediante a execução de casos de teste. O conjunto de casos de teste que exercitou todos esses elementos é considerado adequado ao critério.

Em alguns casos, diz-se que um critério é exigente, ou ainda, que um critério é conservador. Isso significa que tal critério exige mais elementos requeridos, ou requer elementos mais difíceis de serem exercitados do que outros critérios.

A utilização de critérios de seleção/adequação de dados de teste é importante, pois aumenta a probabilidade de encontrar dados mais eficazes, ou seja, que tenham maior capacidade de causar falhas e/ou detectar defeitos.

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

Intuitivamente, bons critérios são aqueles que, se satisfeitos para um dado programa, garantem um bom nível de confiabilidade do mesmo. Portanto, quanto maior a eficácia de um critério, maior a confiança que se pode ter de que, um programa testado segundo esse critério, sem apresentar nenhuma falha, esteja realmente correto.

Naturalmente, existe um compromisso entre o custo e o benefício no processo de teste. Idealmente, os critérios devem requerer um número aceitável de elementos, de forma que o teste de grandes módulos seja factível. Por outro lado, devem fazer com que o testador selecione dados que exercitem o programa de forma tão profunda e completa, que boa parte dos defeitos existentes sejam revelados - idealmente todos os defeitos.