UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE UM SISTEMA GERENCIADOR DE BANCO DE DADOS ORIENTADO A OBJETOS

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

FERNANDO DA SILVA

BLUMENAU, JUNHO/2001.

2001/1-35

PROTÓTIPO DE UM SISTEMA GERENCIADOR DE BANCO DE DADOS ORIENTADO A OBJETOS

FERNANDO DA SILVA

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Marcel Hugo — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Marcel Hugo

_____

Prof. Everaldo Artur Grahl

_____

Prof. Evaristo Baptista

AGRADECIMENTOS

Agradeço ao meu orientador Marcel Hugo, pelo auxílio que me deu durante o desenvolvimento do trabalho.

Aos meus pais, que me apoiaram durante toda a minha vida e sem os quais eu não teria chegado a este momento.

A Deus, por estar presente em todos os momentos e me dando força nos momentos de dificuldade.

Agradeço a meus familiares por tudo que me ensinaram.

A minha namorada, pelo apoio, incentivo, compreensão e carinho que recebi durante a elaboração deste trabalho, principalmente nos momentos difíceis que tive.

A todos os meus colegas de trabalho e de faculdade, que contribuíram para o meu crescimento e tornaram este período de faculdade inesquecível. E também contribuíram e muito na elaboração deste trabalho.

Agradeço também aos Amigos do Barney, que estiveram a meu lado durante toda a faculdade.

Finalmente, agradeço a todos que de alguma forma contribuíram para elaboração deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho tem por objetivo um estudo da aplicação dos conceitos de orientação a objetos como herança, identidade do objeto e tipo abstrato de dados em um banco de dados orientado a objetos, que reúne as aptidões de um banco de dados com os conceitos de orientação a objetos. Para demonstrar estes conceitos, foi desenvolvido um protótipo de um sistema gerenciador de banco de dados orientados a objetos.

ABSTRACT

This paper aims the study of the object oriented concepts application as inheritance, object identification and data abstract type in a object oriented database that meets the capacity of a database and object oriented concepts. In order to present these concepts, a prototype of an object oriented database system manager was developed.

1 INTRODUÇÃO

As tecnologias orientadas a objetos estão cada vez mais presentes em todas as áreas da computação, desde linguagens de programação até os bancos de dados. Através de modelos orientados a objetos pode-se expressar problemas do mundo real de forma mais fácil e naturalmente usando-se componentes modularizados (Khoshafian, 1994).

Esta nova tecnologia gerou um problema, pois os sistemas que utilizam técnicas de orientação a objetos necessitam armazenar suas informações e estas informações estavam sendo armazenadas em bancos de dados relacionais, ou seja, o mundo real estava sendo armazenado em uma coleção de tabelas. Porém, as aplicações como projeto auxiliado por computador (CAD – *Computer-Aided Design*), engenharia de software auxiliada por computador (CASE – *Computer-Aided Software Engineering*), manufatura auxiliada por computador (CAM – *Computer-Aided Manufacture*), bancos de dados multimídia e sistema de informação para escritórios (OIS – *Office Information Systems*) possuem dados mais complexos e que prevalecem mesmo depois de encerrado um determinado processamento. Um banco de dados para uma aplicação CAD armazena informações sobre determinado projeto de engenharia, seus componentes bem como suas antigas versões. Uma complexa rede de objetos é formada em torno de uma estrutura que de forma nenhuma possui tamanho fixo, o que torna praticamente impossível retornar todos os componentes de um projeto em uma única instrução de consulta com uma *Query Language* (Baehr Jr., 1999).

Os bancos de dados orientados a objetos integram os conceitos de orientação a objeto como herança, identidade do objeto e tipo abstrato de dados com as aptidões de um banco de dados como integridade, transação, segurança, recuperação, etc. Através de construções orientadas a objeto, os usuários podem esconder os detalhes de implementação de seus módulos, compartilhar a referência a objetos e expandir seus sistemas através de módulos existentes. As aptidões de banco de dados são necessárias para assegurar o compartilhamento simultâneo e a continuidade das informações nas aplicações (Khoshafian, 1994).

Para demonstrar a aplicação dos conceitos de orientação a objetos foi desenvolvido um protótipo de um sistema gerenciador de banco de dados orientado a objetos. A especificação foi feita utilizando uma metodologia orientada a objetos, representada através

da *Unified Modeling Language* (UML), utilizando como ferramenta para esta especificação o *Rational Rose*, por dar suporte às representações da UML, como o Diagrama de Classes, o Diagrama de Casos de Uso e o Diagrama de Seqüência. A implementação foi desenvolvida no ambiente de programação Borland Delphi 5.0.

1.1 OBJETIVOS

O objetivo do trabalho proposto é demonstrar a aplicação dos conceitos de orientação a objetos como classes, herança, identidade do objeto e tipo abstrato de dados em um banco de dados orientado a objetos através do desenvolvimento de um protótipo de um sistema gerenciador de banco de dados orientado a objetos.

1.2 ESTRUTURA

O presente trabalho está estruturado em cinco capítulos, sendo que no capítulo 1 são apresentados as justificativas, os objetivos e a estrutura do trabalho.

No capítulo 2 são apresentados o histórico, a evolução e considerações gerais sobre sistemas gerenciadores de banco de dados, bem como os conceitos de banco de dados e orientação a objetos aplicados em bancos de dados orientados a objetos.

No capítulo 3 são apresentados alguns conceitos gerais sobre compiladores e árvores B, que foram utilizados no desenvolvimento do protótipo.

No capítulo 4 são apresentados o protótipo, com toda a sua definição, diagrama de classes, caso de uso e diagrama de seqüência, bem como detalhes sobre sua implementação e exemplos sobre a aplicação dos conceitos de orientação a objetos.

No capítulo 5 são apresentadas as considerações finais sobre o trabalho, comentando as conclusões e sugestões para trabalhos futuros.

2 BANCO DE DADOS ORIENTADO A OBJETOS

2.1 HISTÓRICO

Segundo Khoshafian (1994) sistemas de gerenciamento de bancos de dados (SGBD) permitem a bancos de dados persistentes ser concorrentemente partilhados por muitos usuários e aplicativos. Para alcançar isto com eficiência, os SGBD usam controle de concorrência subjacente, manuseio de armazenamento e estratégias de otimização.

Para Date (1991) um SGBD nada mais é do que um sistema de manutenção de dados por computador que tem por objetivo principal manter as informações e disponibilizá-las a seus usuários quando solicitadas.

Segundo Baehr Jr. (1999), junto à estrutura de um banco de dados está o modelo de dados, que é um conjunto de ferramentas para descrição, relacionamento, restrições e semântica de dados. O projeto geral de um banco de dados denomina-se de esquema do banco de dados. Independente do modelo de dados do banco, o bom funcionamento do mesmo está intimamente ligado à definição ideal do esquema de dados. O conjunto de informações existente em um banco de dados em determinado momento denomina-se de instância do banco de dados.

O desenvolvimento dos Sistemas de Gerenciamento de Banco de Dados Orientado a Objetos (SGBDOO) teve origem na combinação de idéias dos modelos de dados tradicionais e de linguagens de programação orientada a objetos.

2.2 EVOLUÇÃO

Os sistemas gerenciadores de arquivos foram os precursores dos SGBD. Eles realizavam funções comuns em arquivos como organização, manutenção e geração de relatórios.

Na década de 60 surgiram os bancos de dados de rede e hierárquico. Esses primeiros modelos de dados eram puramente navegacionais, não tinham forte embasamento teórico e não suportavam a independência física e lógica de dados (Khoshafian, 1994).

Para proporcionar mais flexibilidade na organização de grandes bancos de dados e diminuir alguns problemas dos primeiros modelos, o Dr. E. F. Codd introduziu o modelo de dados relacional no início da década de 70 (Khoshafian, 1994).

Segundo Khoshafian (1994), os SGBD relacionais continuam populares, especialmente em ambientes cliente/servidor.

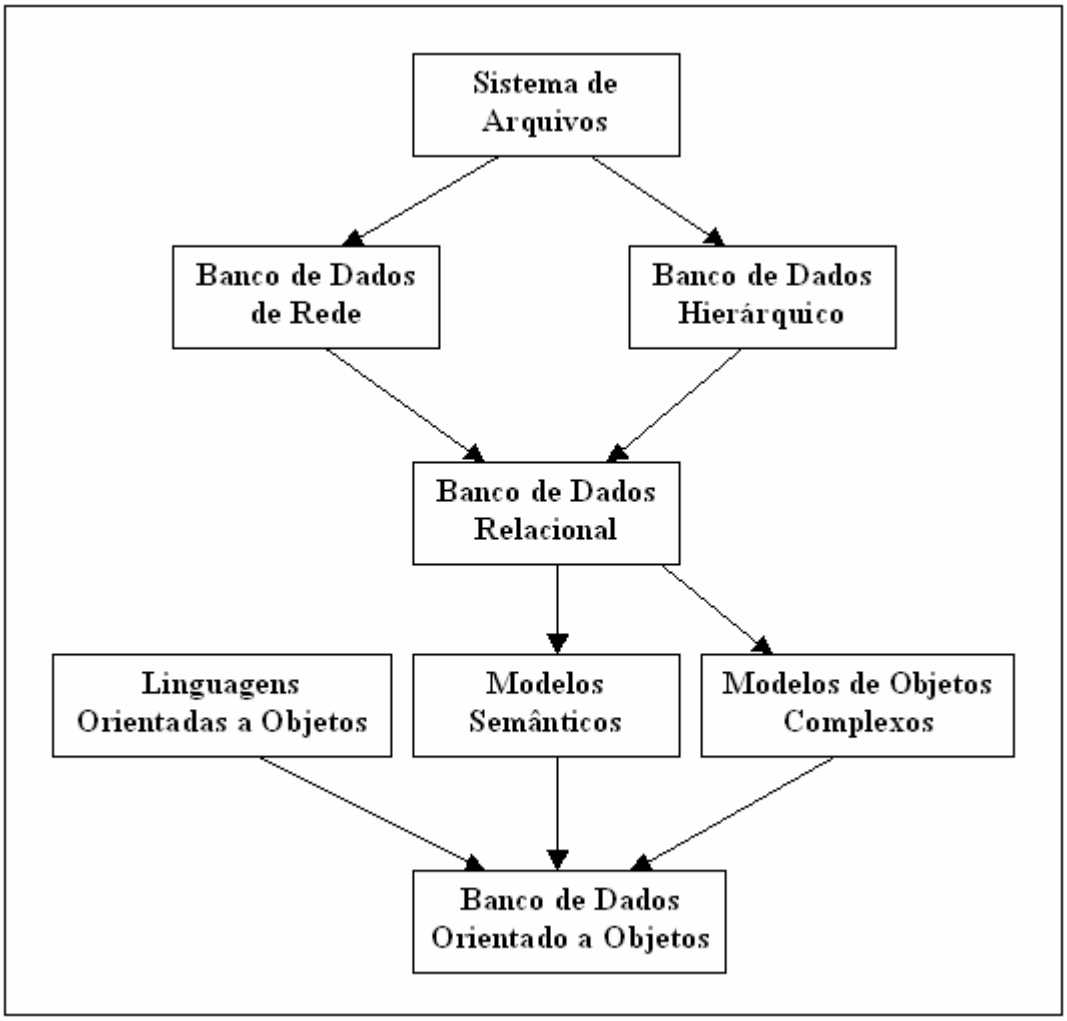
Além dos modelos nos quais os SGBD comerciais se baseavam (relacional, hierárquico e de rede) no final da década de 80 existiam outros modelos como o modelo de dados semântico e o modelo de objetos complexos os quais nunca chegaram a ser comercializados (Baehr Jr. 1999).

Outro fator que influenciou a evolução dos SGBD além do aperfeiçoamento dos modelos de dados, foi a migração na década de 90 dos grandes bancos de dados de *mainframes* para arquiteturas cliente/servidor e adesão, por parte de corporações menores, à tecnologia das redes locais (LAN do inglês *Local Area Networks*). Este processo de ficou conhecido como *downsizing*.

Em meio a este conturbado contexto começam a tomar força os bancos de dados orientados a objetos que haviam surgido em meados da década de 80, que com o *downsizing* e a proliferação dos conceitos de orientação a objetos, tornam-se comercializáveis (Khoshafian, 1994).

A Figura 1 ilustra a evolução dos SGBD até o surgimento dos SGBDOO.

Figura 1 - Evolução dos SGBD



Fonte: Khoshafian (1994).

2.3 CONSIDERAÇÕES GERAIS

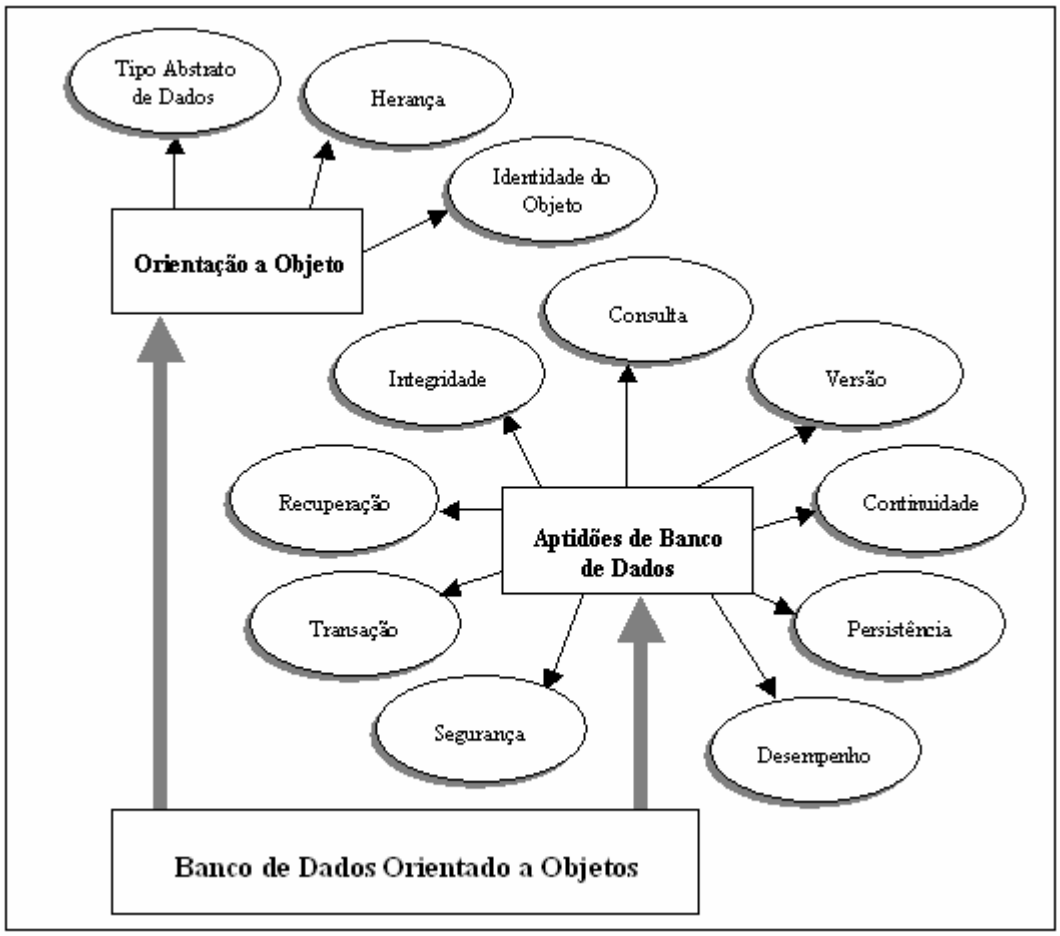
Segundo Khoshafian (1994), os bancos de dados orientados a objeto combinam e integram dois fatores para satisfazer as necessidades de computação: a) a utilização da tecnologia de orientação a objeto que torna fácil construir e manter sistemas complexos de componentes individuais; e b) as aptidões de um servidor de banco de dados que é a medula dos sistemas cliente/servidor, pois serve de depósito de todas as informações compartilhadas na rede.

A modelagem de objetos consiste em “espelhar” o mundo real em objetos que possuem propriedades, comportamentos e eventos que alteram o estado do objeto. Mas segundo Rao

(1994) os bancos de dados convencionais não são flexíveis ou suficientemente eficientes para atender esquemas de dados complexos. Bancos de dados orientados a objetos são sistemas que permitem armazenar um objeto completo como se estivesse na memória principal, sem a necessidade de alterar a estrutura e forma de representação do mesmo.

Pode-se definir um banco de dados orientado a objeto como uma extensão dos conceitos de orientação a objetos como tipagem de dados abstratos, herança e identidade do objeto e aptidões de bancos de dados como continuidade, concomitância, transações, recuperação, filtragem, atualização, integridade, segurança e desempenho como mostra a Figura 2.

Figura 2 - Banco de Dados Orientado a Objeto



Fonte: Khoshafian (1994).

Segundo Baehr Jr. (1999) existem pelo menos três abordagens com relação à construção de SGBDOO. A primeira incorpora aos SGBD convencionais existentes uma

camada responsável por processar as solicitações de Orientação a Objeto (OO) e armazenar os métodos. A segunda defende a inclusão de características de OO nos Sistemas Gerenciadores de Banco de Dados Relacional (SGBDR) existentes, principalmente através da implementação de características como herança e classe na linguagem SQL (*Structured Query Language*). Outra corrente defende a idéia de que é preciso criar uma tecnologia exclusivamente voltada para os bancos de dados orientados a objeto (BDOO) e totalmente desvinculada dos produtos relacionais existentes.

O surgimento dos BDOO está intimamente ligado ao surgimento das aplicações da próxima geração como projeto auxiliado por computador (CAD), manufatura auxiliado por computador (CAM) e escritórios inteligentes, o que inclui automação de escritórios e documentação de imagens (Grein, 1998). Estas aplicações utilizam-se de BDOO por vários motivos, o principal é que os SGBD “convencionais” (Rede, Hierárquico e Relacional) não estão preparados para as necessidades destas aplicações. Nos sistemas de informação para escritórios (OIS), por exemplo, cada objeto do escritório pertence a uma classe que pode ser especializada ou generalizada. O papel do banco de dados nesse caso é armazenar esses objetos e permitir consultas sobre todos os componentes do escritório, como memorandos, documentos, agendas, etc (Baehr Jr, 1999).

2.4 CONCEITOS DE BANCO DE DADOS APLICADOS A SGBDOO

Os conceitos de banco de dados sofrem algumas adaptações quando aplicados a SGBDOO. Estas adaptações são necessárias devido a algumas necessidades dos BDOO. São elas:

a) transações que podem demorar dias para se concretizarem;
b) os objetos devem manter todos os seus níveis disponíveis para acessos concorrentes;
c) as consultas de objetos envolvem estruturas complexas e esbarram no problema da comunicação via mensagens.

Esses processos são implementados de maneira a garantir a integridade e coerência dos objetos (Baehr Jr., 1999).

2.4.1 TRANSAÇÕES

Segundo Date (1991), uma transação é uma unidade de trabalho lógica. Para Khoshafian (1994), uma transação é um programa ou uma seqüência de ações que lê ou grava objetos persistentes e mantém coerente o banco de dados. As transações devem levar o banco de dados de um estado coerente para outro. Para manter esta coerência a transação deve passar pelo teste ACID (Atomicidade, coerência, isolamento, e durabilidade).

a) **atomicidade** – uma transação deve ser executada completamente e com sucesso ou não ser executada (lei do "todo ou nada");
b) **coerência** – um banco de dados é coerente quando todas as restrições de integridade são satisfeitas. Uma transação deve levar o banco de dados de um estado coerente para outro também coerente;
c) **isolamento** – quando as transações são executadas concorrentemente elas devem permanecer isoladas uma das outras, ou seja, uma transação não dever ser influenciada pela execução de outras;
d) **durabilidade** – os resultados de uma transação devem sobreviver a eventuais falhas no sistema ou no meio de armazenamento, estes resultados devem permanecer inalterados até que outra transação os altere e seja finalizado com sucesso.

As transações de aplicações de BDOO são normalmente mais demoradas que as transações de aplicações comerciais convencionais. Várias estratégias foram propostas na pesquisa de banco de dados que tinham relação com a longa duração das transações, algumas destas influenciaram as implementações de BDOO preexistentes (Khoshafian, 1994).

Um dos primeiros modelos de transações proposto foi o modelo de transações aninhadas introduzido por Moss em 1981 (Khoshafian, 1994). Um modelo de transação aninhada pode conter subtransações, também chamadas de transações-filhas. Em uma transação aninhada, todas as transações-filhas devem ser efetivadas para que a transação de alto nível se efetive.

Outro modelo, de transações em cooperação, foi proposto por Nodine em 1992 (Khoshafian, 1994). Este modelo é utilizado quando existe a necessidade de execução de tarefas em conjunto, sendo que estas tarefas podem envolver vários usuários. Neste modelo as transações podem visualizar os resultados umas da outras imediatamente.

2.4.2 CONCORRÊNCIA

O controle de concorrência nada mais é do que coordenar as ações dos processos de diferentes usuários que são executados simultaneamente, acessando dados compartilhados e, portanto, podem interferir nos resultados dos outros.

Existem três estratégias principais para o controle de concorrência: os algoritmos pessimistas que pressupõem que as transações terão conflitos e faz bloqueios antes mesmo de acessar ou atualizar operações; os algoritmos otimistas que pressupõem que as transações não terão conflitos e faz as verificações de integridade da transação somente na hora da efetivação; e os algoritmos de versionamento ou de ordenação de registro de hora no qual é criada uma nova versão de um objeto para cada atualização, com este esquema as transações de leitura sempre serão efetivadas, pois estas podem sempre ler uma versão anterior e consistente do objeto (Khoshafian, 1994).

A maioria dos algoritmos para o controle de concorrência nos SGBD utiliza o bloqueio, que é um algoritmo baseado no pessimismo. Mesmo que outra estratégia (baseada em otimismo ou versionamento) seja utilizada, os BDOO normalmente permitirão que sejam feitos bloqueios explícitos dos objetos. Em um SGBDOO o bloqueio deve ser associado aos vários níveis das estruturas existentes (Khoshafian, 1994).

A idéia do bloqueio é garantir que certo objeto não mude seu estado de forma imprevisível (Date, 1991). Existem vários tipos de bloqueio, os dois mais básicos são o de leitura ou compartilhado e o de gravação ou exclusivo. No bloqueio de leitura outras transações podem realizar concorrentemente operações de leitura, já no bloqueio de gravação qualquer tipo de operação seja ela de leitura ou gravação estão reservadas a transação que fez o bloqueio. Um objeto só pode possuir um bloqueio de gravação de cada vez, ou seja, qualquer outro bloqueio não pode ser disponibilizado, nem mesmo bloqueio de leitura.

Khoshafian (1994) apresenta dois aspectos relevantes para o controle de concorrência em SGBDOO:

a) bloqueios de hierarquia de classe: as classes nos BDOO são organizadas em hierarquias de herança. Sendo assim quando uma superclasse for bloqueada será feito o bloqueio implícito de todas as suas subclasses, que são os descendentes diretos da superclasse e os descendentes das subclasses.

b) bloqueios de objetos complexos: os BDOO podem conter objetos que referenciam ou incorporam outros objetos, isto pode causar problemas, pois um objeto complexo pode possuir alguns de seus subobjetos bloqueados, ao mesmo tempo, na mesma forma que ele foi bloqueado, o que implica em restrições de atualização e possíveis problemas de acesso. Tal problema pode ser resolvido de várias maneiras, mas pode-se salientar o bloqueio de intenção. Neste bloqueio as subclasses de uma classe não são bloqueadas implicitamente, somente as instâncias dependentes do objeto-pai são bloqueadas do mesmo modo que o objeto-pai foi bloqueado. As outras instâncias da classe componente, que não fazem parte do objeto-pai, ficam livres para serem acessadas por outras transações.

2.4.3 RECUPERAÇÃO

Como já havia sido comentado anteriormente, as transações devem ser executadas completamente e com sucesso ou não ser executadas. Para tanto o SGBD deve garantir que as atualizações parciais não sejam efetuadas no banco de dados persistente.

O recurso de recuperação de falhas é muito importante para os SGBD. Segundo Khoshafian (1994) existem três tipos de falhas que o sistema deve se recuperar:

a) falhas de transação: esta falha ocorre quando uma transação não é finalizada com sucesso, ela pode ser causada por problemas de concorrência (*deadlocks*), abortos do usuário ou por violação de alguma restrição de integridade;
b) falhas no sistema: ocorrem por problemas no hardware ou no software, podem ser causadas por erros de software no sistema operacional, no próprio SGBD ou por falta de energia;
c) falhas no meio: ocorrem por problemas no disco rígido ou outros erros irrecuperáveis no disco rígido, são as falhas mais difíceis de restaurar;

Os BDOO utilizam-se de mecanismos como a duplicação ou espelhamento de dados para a gerência de recuperação, mas uma das estruturas mais utilizadas é o *log*. Ele consiste em armazenar as imagens anteriores e posteriores dos objetos atualizados. A imagem anterior é o estado do objeto antes da atualização da transação, e a imagem posterior é o estado do objeto após a atualização da transação.

2.4.4 VERSIONAMENTO

Quando um objeto sofre alteração em um banco de dados existe a necessidade de que sejam guardadas várias versões deste objeto. Aplicações como CAD, CAM e CASE necessitam acessar estados anteriores de determinados objetos. O versionamento nada mais é do que ferramentas e construções de um SGBDOO que automatizam a construção e organização de versões de um mesmo objeto, disponibilizando desta maneira que o usuário acesse versões anteriores do objeto e não somente o seu estado atual.

Segundo Khoshafian (1994) uma configuração pode ser considerada como um grupo de objetos tratados como uma unidade para bloqueio e versionamento. Os objetos individuais dentro da configuração podem sofrer modificações, de modo que cada objeto pode ter um histórico das versões. Vários objetos dentro da configuração são atualizados em momentos diferentes e não necessariamente na mesma freqüência.

2.4.5 RESTRIÇÕES DE INTEGRIDADE

Segundo Korth (1995), as restrições fornecem meios para assegurar que mudanças feitas no banco de dados por usuários autorizados não resultem na perda da consistência dos dados.

Os SGBD tradicionais possuem vários tipos de restrições de integridade. A maioria destas restrições se aplica aos SGBDOO também, mas devido as diferentes construções e o poder do modelo de dados OO, alguns dos tipos deixam de ser relevantes, outros tomam formas diferentes e alguns novos tipos de restrições são introduzidos nos BDOO.

2.4.5.1 RESTRIÇÕES DE CHAVE

Nos bancos de dados relacionais (BDR) é utilizada a especificação de chaves para as tabelas para identificar exclusivamente os registros de uma tabela tanto em nível lógico quanto em nível físico e também para otimizar os processos de consulta. Embora a função de identificar exclusivamente as instâncias de uma classe em nível físico nos BDOO seja do identificador do objeto (OID – *Object Identifier*), os BDOO permitem que sejam definidas chaves para uma determinada classe, para identificar a mesma em nível lógico e otimizar os processos de consulta (Baehr Jr, 1999).

2.4.5.2 RESTRIÇÃO REFERENCIAL

Segundo Baehr Jr. (1999), os objetos de um banco de dados normalmente estão relacionados com outros objetos no banco de dados, por este motivo surge a necessidade da integridade referencial, para assegurar que um objeto não esteja relacionado com um objeto que não exista no banco de dados.

O mecanismo de chave estrangeira é o mais utilizado para referenciar objetos entre si nos modelos baseados em valor. Ele relaciona os objetos através dos valores de seus atributos, sendo que o valor de um atributo em um objeto é a chave de um outro objeto. Os SGBDOO não necessitam do mecanismo de chave estrangeira, pois o conceito de OID permite referenciar objetos diretamente (Khoshafian, 1994).

2.4.5.3 RESTRIÇÃO EXISTENCIAL

Esta restrição somente é suportada por sistemas que implementem o conceito de OID. Seu objetivo é assegurar que um objeto compartilhado referencialmente possua um domínio "ativo" no qual ele exista. Ou seja, quando for informado um valor para um determinado atributo de um objeto, este valor deve existir em um outro grupo específico de objetos.

2.4.5.4 RESTRIÇÕES *NOT NULL*

Para que um SGBD possa suportar restrições *NOT NULL* ele deve suportar valores *NULL*. Um valor *NULL* significa que este valor não existe ou está indisponível. Quando um atributo ou coluna é declarado como do tipo T, os valores que este atributo pode receber são valores T ou *NULL*. Quando uma restrição *NOT NULL* é definida para um determinado atributo, este passa a não aceitar o valor *NULL* como um valor válido, são aceitos somente valores correspondentes à faixa de valores definidas para o atributo (Khoshafian, 1994).

2.4.5.5 REGRAS DE INTEGRIDADE

Segundo Khoshafian (1994), existem alguns tipos de restrições que não são contempladas pelas restrições de integridade especiais que foram vistas até agora. Para estas é necessário um mecanismo geral através do qual os usuários possam expressar suas próprias regras de integridade. A diferença entre as restrições é que as restrições de integridade especiais são suportadas pelo sistema, enquanto o mecanismo de restrição de integridade geral

é definido pelo usuário. No futuro os sistemas provavelmente irão suportar ambos os mecanismos.

2.4.5.6 GATILHOS (*TRIGGERS*)

Os gatilhos são um pouco diferentes das restrições vistas até aqui, pois quando uma restrição de integridade é violada, o sistema levanta uma exceção e normalmente a resposta a esta exceção é abortar a transação. Já os gatilhos são ativados quando determinadas operações são executadas ou uma condição é satisfeita.

Para Korth (1995), um gatilho é um comando executado automaticamente pelo sistema como um efeito colateral de uma modificação no banco de dados. Segundo Khoshafian (1994), o mecanismo de gatilho permite ao desenvolvedor de aplicações de bancos de dados garantir que restrições de integridade sejam mantidas por meio da ação que é ativada na condição.

2.4.5.7 PRÉ-CONDIÇÕES E PÓS-CONDIÇÕES PARA MÉTODOS

Para Khoshafian (1994), pré-condições e pós-condições proporcionam um método eficiente para depuração e suporte de restrições de integridade.

Pré-condições permitem que sejam definidas certas restrições nas variáveis de instância que devem ser satisfeitas para que um determinado método possa ser executado. Pós-condições permitem que sejam definidas outras restrições que devem ser satisfeitas ao término da execução do método.

2.4.5.8 RESTRIÇÃO DE DISJUNÇÃO

A restrição de disjunção determina que duas classes não podem ter o mesmo elemento em comum, ou seja, se duas classes forem disjuntas o sistema não irá permitir que seja criada uma subclasse que herde as duas classes ao mesmo tempo (Khoshafian, 1994).

2.4.5.9 RESTRIÇÃO DE COBERTURA

A restrição de cobertura determina que uma superclasse não pode possuir instâncias que não sejam elementos de um dado grupo de subclasses da mesma. Uma maneira de obter

esta restrição é utilizando classe abstrata, desta maneira nunca seria possível criar uma instância da superclasse, somente de suas subclasses. Mas nem sempre este meio é o ideal, pois podem existir situações onde seja necessário instanciar uma superclasse, mas mesmo assim manter a restrição de cobertura, sendo que o sistema iria levantar uma exceção somente se esta instância criada fosse salva como um elemento da superclasse e não como um elemento de uma das subclasses de cobertura (Khoshafian, 1994).

2.4.6 PERSISTÊNCIA

Para Nassu (1999), a característica que diferencia um BDOO em relação às linguagens de programação orientadas a objetos (LPOO) é a persistência dos objetos. Persistência é a capacidade dos objetos de continuarem existindo mesmo após o encerramento de um programa ou transação, sendo que seu estado é armazenado em um meio físico persistente.

Existem basicamente três formas de tornar o objeto persistente, a) por tipo (classe) onde os objetos que pertencem a uma classe declarada como persistente serão persistentes, b) por chamada explícita quando na criação dos objetos um comando especial torna-os persistentes e c) por referência onde os objetos originados de objetos persistentes tornam-se persistentes. As duas últimas formas são as mais preferidas, pois segundo Khoshafian (1994) a persistência de objetos deve ser ortogonal ao tipo do objeto, ou seja, qualquer objeto pode ser persistente, dependendo da necessidade da aplicação.

Uma das grandes vantagens da persistência é que as aplicações não precisam mais se preocupar com a conversão de um objeto da memória para um meio físico permanente. Esta conversão é uma responsabilidade do SGBDOO.

2.4.7 CONSULTAS

Nas LPOO é necessário que a interação com os objetos seja feita através de mensagens, o que representa uma grande limitação para bancos de dados, pois caso fosse necessário consultar a ocorrência de um determinado atributo nas instâncias de um objeto seria necessário enviar uma mensagem para cada instância do mesmo. Para solucionar este problema existe nos BDOO um modelo de mensagem de objeto que permite que sejam formuladas consultas que envolvam junções de conjuntos de objetos.

Segundo Korth (1995), uma linguagem de consulta para um SGBDOO precisa possuir tanto o modelo de passagem de mensagens para um único objeto ou para um conjunto.

2.5 CONCEITOS DE ORIENTAÇÃO A OBJETOS APLICADOS A SGBDOO

Segundo Baehr Jr. (1999), a noção de objetos nos SGBDOO é utilizada em nível lógico e possui características que não são encontradas em LPOO, como operadores de manipulação de estruturas, gerenciamento de armazenamento, tratamento de integridade e persistência de objetos. Dessa forma, assim como os conceitos de banco de dados os conceitos de OO também sofrem algumas adaptações quando aplicados em SGBDOO.

Para Nassu (1999), um banco de dados pode ser considerado orientado a objetos quando ele possui as seguintes características: presença de identificadores de objetos, mecanismos de herança, métodos, objetos complexos e persistência de objetos. Estes e outros conceitos de OO serão demonstrados a seguir.

2.5.1 TIPOS DE DADOS ABSTRATOS

Os tipos de dados abstratos não especificam somente uma estrutura de objeto como os tipos de dados normais. Eles especificam também o seu comportamento, ou seja, eles proporcionam um mecanismo adicional aos tipos de dados pelo qual é feita uma clara separação entre a interface e a implementação dos tipos de dados.

Os tipos de dados abstratos possuem operações comuns que são utilizadas para manipular as estruturas de dados das suas instâncias. Além disso, todas as manipulações de instâncias de um tipo de dados são realizadas exclusivamente por operações associadas àquele tipo de dados (Khoshafian, 1994).

2.5.2 CLASSES

Segundo Khoshafian (1994), classe é a construção mais utilizada para definir um tipo de dados abstrato, pois ela incorpora tanto a estrutura quanto as operações do tipo de dados abstratos.

Para Nassu (1999), uma classe é um modelo de onde os objetos são criados (instanciados). Os objetos de uma mesma classe possuem a mesma estrutura e comportamento.

Já Rumbaugh (1994), define classe como sendo um grupo de objetos com propriedades semelhantes, o mesmo comportamento, os mesmos relacionamentos com outros objetos e a mesma semântica.

2.5.3 HERANÇA

Herança é um mecanismo que permite ao usuário definir novas classes baseadas nas classes já existentes. A herança permite a reutilização de propriedades para a definição de uma nova classe, esta subclasse herda todas as características de seus ancestrais e também acrescenta seus próprios atributos e operações, ou seja, ela corresponde à transferência de propriedades estruturais e de comportamento de uma classe para suas subclasses (Rumbaugh, 1994).

A herança pode ser simples (única) quando a classe herda características de apenas uma superclasse ou múltipla quando a classe herda características de duas ou mais superclasses.

As principais vantagens de herança são prover uma maior expressividade na modelagem dos dados, facilitar a reusabilidade de objetos e definir classes por refinamento, podendo fatorar especificações e implementações como na adaptação de métodos gerais para casos particulares, redefinindo-os para estes, e simplificando a evolução e a reusabilidade de esquemas de banco de dados.

2.5.4 IDENTIDADE DE OBJETO

A impossibilidade de garantir a identificação de objetos exclusivamente através de sua estrutura e comportamento, foi o que motivou a definição de identificadores únicos de objetos, que persistem no tempo de forma independente ao estado interno do objeto (Korth, 1995).

O identificador de objeto (OID) é o que distingue um objeto dos demais. Nos BDOO, os objetos possuem identidade mais forte que nas LPOO, pois eles devem continuar existindo mesmo após a execução do programa ou transação e podem voltar ser utilizados, na próxima execução, ou mesmo por outros programas ou transações ao mesmo tempo.

O OID deve ser único e imutável, desde a sua criação até a eliminação física do objeto na base de dados. A sua existência é independente de seus valores ou do seu endereço de armazenamento físico. A identidade do objeto é geralmente gerada pelo sistema quando o objeto é criado.

Sendo assim, segundo Baehr Jr. (1999) um OID em SGBDOO não pode ser:

a) um endereço porque não é externo ao objeto;
b) uma chave porque não é um valor de dados mutável;
c) um *register* ID (como o *ROWID* do *Oracle*) porque não é uma coluna lógica.

Por serem identificadores únicos, os OIDs também são utilizados para estabelecer o relacionamento entre os objetos no BDOO e servem como uma forma de se recuperar os objetos do banco de dados. O usuário não deve ter acesso ao valor do OID, nem mudar o seu valor (Nassu, 1999).

A identidade de objetos elimina as anomalias de atualização e de integridade referencial, uma vez que a atualização de um objeto será automaticamente refletida nos objetos que o referenciam e que o identificador de um objeto não tem seu valor alterado.

2.5.5 OBJETOS COMPLEXOS

Segundo Martin (1995), objetos complexos são objetos que contém como algum atributo seu uma referência a outro objeto. Dessa forma, objetos podem ser compostos de objetos que por sua vez também podem ser compostos de objetos e assim sucessivamente.

Os objetos complexos são formados por construtores (conjuntos, listas, tuplas, registros, coleções, *arrays*) aplicados a objetos simples (inteiros, *booleanos*, *strings*). Nos modelos orientados a objetos, os construtores são em geral ortogonais, isto é, qualquer construtor pode ser aplicado a qualquer objeto.

Os SGBDOO devem oferecer mecanismos para se recuperar por completo um objeto complexo do dispositivo de armazenamento para a memória, e também manter íntegro os seus relacionamentos (Baehr Jr., 1999).

Existem dois tipos básicos de objetos complexos em BDOO:

a) objetos embutidos: são “objetos filhos”, na forma de atributos, que só podem ser acessados pelo seu “objeto-pai”, estes objetos não possuem OID próprio e geralmente são armazenados na mesma estrutura física do “objeto-pai”;
b) objetos referenciados: são todos os objetos originários das regras de integridade referencial, estes objetos possuem OID próprio e podem ser acessados diretamente ou através de seus objetos relacionados.

A manutenção de objetos complexos, independente de sua composição, requer a definição de operadores apropriados para sua manipulação como um todo, e transitivos para seus componentes. Exemplos destas operações são: a atualização ou remoção de um objeto.

2.5.6 EXTENSIBILIDADE

Para Grein (1998) a extensibilidade é uma propriedade fundamental em um SGBDOO. Esta propriedade garante que os novos tipos de dados criados pelos usuários não podem ter diferenças no tratamento em relação aos tipos pré-definidos pelo sistema, pelo menos na visão do usuário.

2.5.7 COMPLETUDE COMPUTACIONAL

Assim como a extensibilidade a completude computacional é uma propriedade fundamental em um SGBDOO, ela implica em que a linguagem de manipulação de um banco de dados orientado a objetos possa exprimir qualquer função computacional (Grein, 1998) (Nassu, 1999).

2.6 BANCOS DE DADOS CONHECIDOS

Existem atualmente algumas implementações de BDOO. Alguns são protótipos de empresas, universidades e até produtos comerciais. Dentre eles pode-se citar o O_2, Objectstore, Jasmine, Postgres, Caché entre outros. A seguir serão apresentadas as principais

características destes sistemas. Será dado destaque para a linguagem de definição de manipulação de dados.

As sintaxes das linguagens de definição e manipulação de dados dos bancos O_2, Objectstore, Jasmine e Postgres, foram retiradas de Nassu (1999), e segundo o mesmo nem todas puderam ser testadas, portanto podem não estar 100% corretas. Já a sintaxe utilizada pelo banco de dados Caché foi retirada de Baehr Jr. (1999).

2.6.1 O_2

O projeto do sistema foi iniciado na França em 1988. Passou a ser comercializado pela O_2 Technology a partir de 1991, com o fim do convênio que o financiava.

A declaração dos dados é feita através de uma linguagem que é uma extensão da linguagem C, o O_2C. A linguagem possui herança múltipla. Existem também construtores de tipos complexos de dados e não há diferença na declaração de objetos persistentes e não persistentes (Nassu, 1999).

O Quadro 1 mostra um exemplo da declaração de uma classe Pessoa no O_2.

Quadro 1 - Declaração de uma classe Pessoa no O_2

```
Class Pessoa
   Type tuple(nome: string
              RG: string
              Endereço: string)

   Method Muda_Endereco(novoEndereco: string)
end
```

Fonte: Nassu (1999).

O O_2 garante automaticamente a integridade referencial nos relacionamentos. Se um dos objetos for excluído do banco de dados, o sistema automaticamente torna as referências a este objeto nulas.

A manipulação dos dados pode ser feita de duas maneiras: através da sua linguagem de programação, ou pode-se fazer consultas com o OQL que é uma linguagem de consulta do tipo SQL. Segundo Nassu (1999), para fazer um objeto tornar-se persistente, usa-se o comando *add name*, que faz com que o OID de um objeto seja associado a um nome de uma

variável, que posteriormente pode ser recuperado. Outra forma de tornar persistente o objeto é fazer com que ele seja referenciado por outro objeto que já é persistente. O Quadro 2 demonstra como é feita a persistência e a manipulação de dados no O_2.

Quadro 2 - Persistência e manipulação de dados no O2

```
/* Cria objeto aluno persistente */
add name Eugenio: Aluno;
/* Cria conjunto de alunos persistente */
add name AlunosDoBCC: set(Aluno);
/* Cria um aluno, a princípio, não persistente */
Francisco: Aluno;

run body
{
Eugenio.Nome = "Eugênio Akihiro Nassu";
/* modificando/iniciando o objeto Aluno */
...
AlunosDoBCC += set(Eugenio);
/* colocando o Aluno em um conjunto */
AlunosDoBCC += set(Francisco);
/* o aluno Francisco também se torna persistente
pois é referenciado por um objeto persistente */
...
/* modifica o endereço do Aluno Eugenio */
Eugenio.Muda_Endereco("Alameda Barros 380");
...
}

select tuple(Aluno: m.Al.Nome, Nota: m.nota)
from m in Matriculas
where m.Disc.nome = "MAC-110" and m.SemAno = "1/95"
```

Fonte: Nassu (1999).

2.6.2 OBJECTSTORE

Este banco de dados é produzido pela Object Design Inc. seu principal objetivo é tornar a linguagem C++ uma linguagem para banco de dados. A principal característica do sistema é o tratamento uniforme dos objetos comuns e persistentes, sem que haja perda de performance. O Objectstore utiliza um mecanismo onde os objetos persistentes fazem parte da memória virtual, ou seja, ficam gravados em disco. Quando se faz um acesso ao objeto, é gerada uma falha de página de memória no sistema, que faz com que o objeto seja carregado para a memória principal.

Por utilizar os tipos e a linguagem C++ como base, as declarações são quase idênticas, fazendo com que a conversão de programa C++ para Objectstore necessite apenas de pequenas modificações (Nassu, 1999).

O Quadro 3 demonstra a um exemplo de declaração de uma classe no ObjectStore.

Quadro 3 – Exemplo de declaração de classe no ObjectStore

```
class Pessoa {
  char *nome;
  char *RG;
  char *endereço;

  void Muda_Endereco( char *NovoEndereco );
}
```

Fonte: Nassu (1999).

Para manipular e persistir os dados é necessário declarar uma variável do tipo *database*. O Quadro 4 mostra um exemplo da utilização da variável *database*.

Quadro 4 – Exemplo da utilização da variável *database*

```
#include <objectstore/objectstore.H>
#include <records.H>

main(){

  //declara uma variável bando de dados
  database *db;

  //abre um banco de dados
  db = database::open("dados/IME");

  //inicia transação
  transaction::begin();
  //cria conjunto persistente
  os_set <Aluno *> Alunos = os_set(Aluno *)::create(db);
  .
  .
  .
  //cria novo objeto. Já é criado como persistente, pela extensão do
  //operador new
  aluno a = new(db) aluno ("Eugênio");
  //insere objeto no conjunto
  Alunos->insert(a);
  //grava as alterações em definitivo
  transaction::commit();
}
```

Fonte: Nassu (1999).

2.6.3 JASMINE

O Jasmine foi desenvolvido pela Fujitsu, do Japão. Inicialmente o gerenciador de objetos foi implementado como uma “casca” de um banco de dados relacional não-

normalizado, da mesma empresa. A empresa que atualmente comercializa o sistema é a Computer Associates. Pode-se dizer que é o primeiro BDOO a ser comercializado por uma grande empresa de software (Nassu, 1999).

O Jasmine utiliza a linguagem ODQL (*Object Database Query Language*) para a definição e manipulação de dados.

O sistema permite a criação de famílias de classes e herança múltipla. Sempre que uma classe é criada é necessário que se informe a família que a mesma pertence. Somente é exigida a unicidade do nome das classes dentro de cada família. O Quadro 5 demonstra a definição de classes do Jasmine.

Quadro 5 – Exemplo de definição de classes no Jasmine

```
DefineClass demoCF::Pessoas
  description: "informação sobre uma pessoa"
{
  maxInstanceSize: 4;
  class:
    Integer   proxNumPessoa default:0;
  Instance:
    Integer     NumPessoa   unique;
    String      Nome        mandatory;
    Date        DataNasc;
    Bag<string> Telefones   default:Bag{};
    Integer     Idade();
}
```

Fonte: Nassu (1999).

As consultas retornam valores que normalmente são atribuídos a uma variável, o que lembra o “cursor” da linguagem SQL. Através destas variáveis é feita a manipulação dos dados. O Quadro 6 mostra uma seqüência de comandos de manipulação de dados no Jasmine.

Quadro 6 – Exemplo de comandos de manipulação de dados no Jasmine

```
//Cria um novo objeto
Pessoas p;
p = Pessoas.new( Nome := "Romário" );

//Modifica um objeto existente
Pessoas pp;
pp = Pessoas from Pessoas where Pessoas.NumPessoa = 111
pp.Nome = "Novo Nome!"

//Exclui um objeto existente
Pessoas pp;
pp = Pessoas from Pessoas where Pessoas.NumPessoa = 222
pp.delete();
```

Fonte: Nassu (1999).

2.6.4 POSTGRES

O Postgres é um protótipo desenvolvido na Universidade da Califórnia, Berkeley. Ele foi desenvolvido para ser o sucessor do Ingres que é um banco de dados relacional, que foi desenvolvido pela mesma universidade.

O sistema é baseado num modelo relacional estendido, suportando objetos, OIDs, objetos complexos, herança múltipla, versões, dados históricos, e uma linguagem de consulta o PostQuel, que é uma extensão da linguagem do Ingres, QUEL (Nassu, 1999).

A possibilidade de armazenar dados históricos é uma das características mais importantes do Postgres. Através dele é possível consultar o estado do banco de dados em um determinado momento do passado.

Segundo Nassu (1999), o Postgres tem um modelo de dados baseado no modelo relacional. O sistema oferece um tipo abstrato de dados, para que se possa definir um novo tipo de dado. O Quadro 7 mostra um exemplo de definição deste tipo abstrato de dados.

Quadro 7 – Exemplo de definição de um tipo abstrato de dados no PostGres

```
create pessoa( nome = char[25],
               Endereco = char[30],
               RG = char[15]
key(RG) )
```

Fonte: Nassu (1999).

O Quadro 8 mostra um exemplo de atribuição de valores para um objeto.

Quadro 8 – Exemplo de atribuição de valores para um objeto

```
append to pessoa (
   nome = "Eugênio Akihiro Nassu",
   endereço = "Rua São Pedro 444",
   RG = "12345678"
)
```

Fonte: Nassu (1999).

2.6.5 CACHÉ

O banco de dados é fornecido pela InterSystems Corporation. Segundo Baehr Jr. (1999), a empresa é considerada uma das 10 maiores distribuidoras de bancos de dados pós-relacionais (bancos de dados com modelos de dados que surgiram após o modelo de dados relacional (Khoshafian, 1994)) para sistemas UNIX no mundo.

O Caché é um banco de dados leve e rápido. Ele é livre de declarações o que significa que não é necessário declarar o número de dimensões e os limites de valores dos dados no modelo. E totalmente dinâmico, pois não é necessário se preocupar com a administração de espaço em memória ou em disco durante a eliminação e adição de registros e atributos (Baehr Jr., 1999).

Existem três formas de acesso aos dados do Caché: direto, SQL ou por Objeto. Segundo Baehr Jr. (1999) dessa forma, ferramentas RAD (do inglês *Rapid Application Development* – Desenvolvimento Rápido de Aplicativos), como o Borland Delphi e o Microsoft Visual Basic ou linguagens como C++ e Java podem utilizar o acesso via tabelas tradicional, enriquecer seus modelos de dados como os modelos de objetos ou otimizar seus processos com o acesso direto.

O Caché possui em sua estrutura um componente chamado Caché Objects, que é responsável por disponibilizar simultaneamente, o desempenho e o poder de modelagem da estrutura multidimensional de dados do Caché associada às características da tecnologia OO. Ele é ao mesmo tempo uma LPOO e um SGBD que “rodam” em cima da estrutura de dados do banco Caché.

O Caché Objects oferece suporte a herança, inclusive herança múltipla. Existem duas formas de definir classes:

a) através do Caché Object Architect, uma ferramenta com interface gráfica para o usuário que serve para otimizar o processo de criação e manipulação de classes;
b) através do Caché Class Description Language (CDL), uma linguagem de definição de classes apresentada em um arquivo texto no formato ASCII.

O Quadro 9 mostra um exemplo de criação da classe Pessoa em CDL.

Quadro 9 – Definição de classe em CDL

```
create class Pessoa
{
  description = "Uma simples classe de exemplo";
  final;
  super = %RegisteredObject,%Persistent;
  persistent;

  parameter KEYS           {default = "CPF";}
  parameter INDEX          {default = "Nome";}

  attribute Nome           {type = %String(MAXLEN=35); required;}
  attribute CPF            {type = %String(MAXLEN=12); required;}
  attribute Telefone       {type = %String(MAXLEN=12); required;
                                initial = "(000) 000-0000";}

  method editCPF()
  {
    returntype = %string;
    description = "Um Exemplo de Método";
    final;
    public;
    code = {
    Quit$E(..CPF,1,3)_"."_$E(..CPF,4,6)_"."_$E(..CPF,7,9)_"."_$E(..CPF,10,11)
          }
  {
}
```

Fonte: Baehr Jr. (1999).

3 CONCEITOS GERAIS

Neste capítulo são descritos alguns conceitos e termos utilizados durante a descrição do protótipo.

O conhecimento sobre os conceitos de compiladores é necessário, pois o protótipo irá utilizá-los quando fizer a interpretação e execução da linguagem de definição de dados e da linguagem de manipulação de dados.

Já a árvore B é utilizada na organização física dos dados no banco de dados. É através de uma árvore que é feita a indexação, busca, inclusão e retirada dos dados.

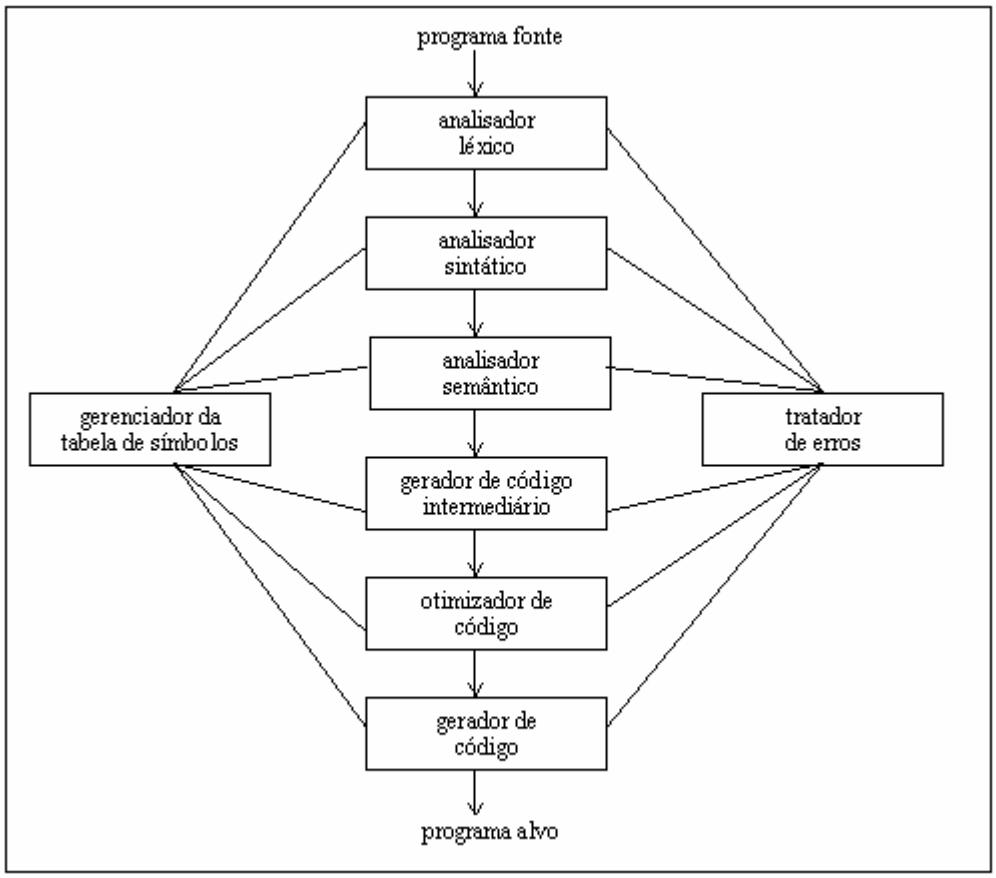
3.1 COMPILADOR

Segundo Aho (1995), um compilador é um programa que lê um programa escrito numa linguagem (linguagem fonte) e o traduz num programa equivalente numa outra linguagem (linguagem alvo).

Setzer (1985) define um compilador como um programa que tem a finalidade de converter um programa escrito em uma linguagem-fonte para um programa escrito em outra linguagem-objeto.

O compilador opera em fases, e em cada uma delas transforma o programa fonte de uma representação para outra. Uma decomposição típica de um compilador é mostrada na Figura 3.

Figura 3 – Fases de um compilador



Fonte: Aho (1995).

Segundo Setzer (1985), as principais vantagens de construir um compilador em fases são as seguintes:

a) menor utilização de memória do computador, já que cada passo exerce apenas uma parte das funções de todo o compilador;
b) maior possibilidade de se efetuar otimizações levando a um programa-objeto mais eficiente quanto ao espaço ocupado ou ao tempo de processamento;
c) os objetos e implementações das várias partes do compilador são mais independentes.

As principais desvantagens são:

a) maior volume de entrada/saída, quando os programas intermediários e os passos do compilador não ficam residentes na memória, o que é em geral o caso;
b) normalmente, aumento do tempo de compilação;

c) aumento do projeto total, com a necessidade de introdução de linguagens intermediárias.

A seguir serão descritas as três fases que serão utilizadas no protótipo, análise léxica, análise sintática e análise semântica. Segundo Aho (1995) estas três fases formam o núcleo da parte de análise do compilador.

3.1.1 ANÁLISE LÉXICA

Segundo Aho (1995), a análise léxica é a primeira fase de um compilador. A tarefa principal do analisador é ler os caracteres de entrada e produzir uma seqüência de *tokens* que o *parser* utiliza para a análise sintática.

Para Setzer (1985), a função do analisador léxico é varrer o programa fonte da esquerda para a direita, agrupando os símbolos em cada item léxico (*token*) e determinando sua classificação.

Os *tokens* são “palavras” as quais o analisador léxico deve classificar. Entre elas pode-se citar: palavras-chave, operadores, identificadores, constantes, literais, cadeias e símbolos de pontuação, como parêntesis, vírgulas e ponte-e-vírgulas.

3.1.2 ANÁLISE SINTÁTICA

A análise sintática tem a função de determinar qual a ordem correta dos *tokens* na frase. Para Aho (1995), o papel do analisador sintático é obter a cadeia de *tokens* geradas pelo analisador léxico e verificar se esta cadeia pode ser gerada pela gramática definida, ou seja, se não possui nenhum erro sintático.

Caso exista algum erro sintático o analisador deve emitir a mensagem de erro adequada, tratar o erro e continuar a análise do programa, de maneira que o erro ocorrido influencie o mínimo possível a análise do restante (Setzer, 1985).

3.1.3 ANÁLISE SEMÂNTICA

A fase análise semântica é responsável pela verificação dos erros semânticos e armazenagem de informações de tipo para que possa ser realizada a geração de código.

Segundo Aho (1995), um importante componente da análise semântica é a verificação de tipos. Nela o compilador checa se cada operador recebe os operandos que são permitidos pela especificação da linguagem.

3.2 ÁRVORE B

Um dos exemplos mais significativos de estruturas de dados não-lineares que representa relações de hierarquia e composição é a árvore (Villas, 1993). Uma árvore pode ser definida como sendo um conjunto finito, de um ou mais nós, tais que:

a) existe um nó denominado raiz da árvore;
b) os demais nós formam M >= 0 subconjuntos disjuntos S1, S2,...Sm, onde cada um desses subconjuntos é uma árvore. As árvores Si (1 =< i =< M) recebem a denominação de subárvores.

Existem alguns tipos de árvore, mas segundo Villas (1993), quando se deseja manipular informações na memória secundária é interessante que se agrupe várias chaves em um único nó, pois o tempo de acesso a um nó de uma árvore em disco é muito maior do que se ele estivesse em memória. Desta forma reduz-se o número de acessos à estrutura, e conseqüentemente ao disco, diminuindo assim o tempo de pesquisa. Além disso, esse agrupamento faz com que a altura seja reduzida.

Este tipo de árvore, onde são agrupadas as chaves é chamada de árvore de busca M-vias ou multivias (*multiway tree*). Essa estrutura é uma árvore de grau M.

Segundo Villas (1993), para que a estrutura possua um algoritmo de pesquisa eficiente, deve-se considerar não somente o tempo de acesso a cada nó, mas também o balanceamento da árvore. Uma árvore é dita balanceada quando, para qualquer nó, as suas subárvores à esquerda ou à direita possuem a mesma altura. As árvores de M-vias que por definição estão permanentemente balanceadas são chamadas Árvores B.

Uma árvore B de ordem d, segundo Villas (1993), é uma árvore de busca M-vias balanceada com as seguintes características:

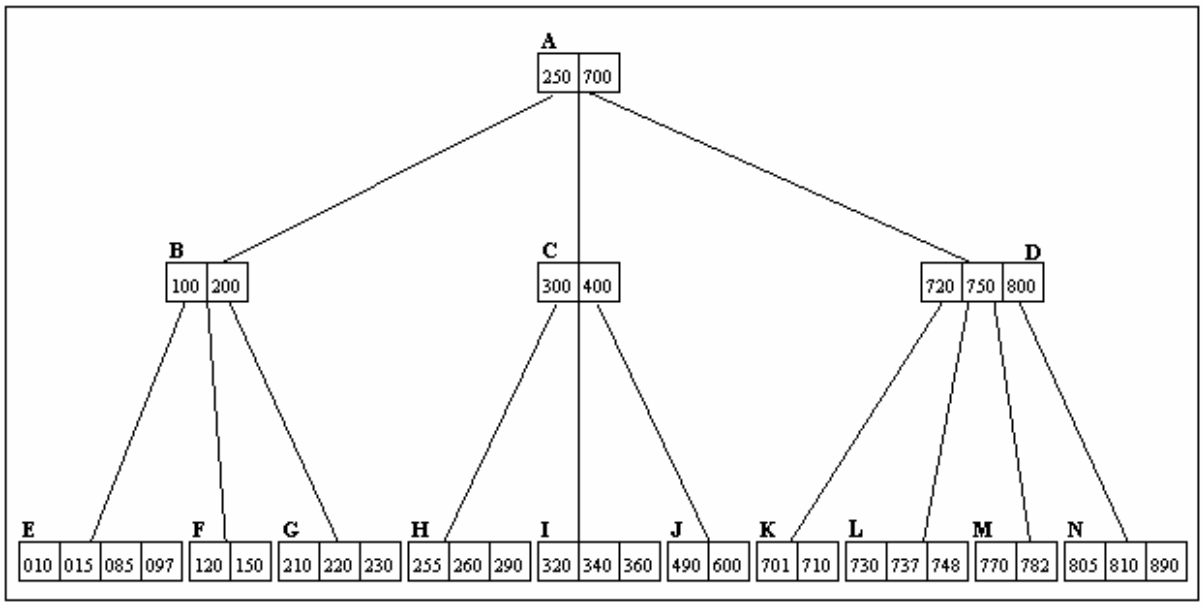
a) a raiz tem no mínimo uma chave e dois filhos;
b) uma folha tem no mínimo d chaves e não tem filhos;
c) todos os outros nós tem no mínimo d chaves e d + 1 filhos;

d) todos os nós têm, no máximo, 2d chaves e 2d + 1 filhos;

e) todas as folhas estão no mesmo nível (balanceamento).

A ordem d determina as quantidades máxima e mínima de chaves dentro de cada nó da árvore B. O número máximo de chaves é estabelecido de acordo com o espaço físico disponível para cada nó. O número mínimo de chaves é estabelecido para determinar o percentual mínimo de ocupação dentro de um nó e, portanto, da estrutura. Na árvore B esse percentual é de 50% (não considerando a raiz) (Villas, 1993). A Figura 4 mostra uma árvore B de ordem 2.

Figura 4 – Árvore B de ordem 2



Fonte: Villas (1993).

4 PROTOTIPAÇÃO

O protótipo implementado neste trabalho é um sistema gerenciador de banco de dados orientado a objetos, no qual são demonstrados os seguintes conceitos de orientação a objetos aplicados em um SGBDOO: classes, herança, identidade do objeto e tipo abstrato de dados.

O protótipo opera pela execução de *scripts* através dos quais permite que seja definido um conjunto de classes, nas quais poderão ser identificados os conceitos descritos acima. A definição será verificada obedecendo a uma linguagem de definição de dados (LDD). Esta definição irá gerar um dicionário de dados que conterá a definição das classes seus respectivos métodos e atributos e ligações, e também as propriedades de cada método e atributo.

Para especificar a sintaxe da LDD do protótipo foi utilizada uma notação que segundo Aho (1995) é amplamente aceita, que é a BNF (Forma de Backus-Naur), sendo uma gramática livre de contexto.

O Quadro 10 demonstra a LDD definida onde as palavras reservadas e os símbolos especiais estão sublinhados. Os não terminais classificados pelo analisador léxico estão em negrito. Superclasse define qual é a classe mãe da classe que está sendo definida, ou seja, a classe de onde a classe é derivada. O que indica que uma classe pode persistir suas instâncias é a propriedade “persistente”.

Quadro 10 - Definição da Linguagem de Definição de Dados (LDD)

```
<BD>              ::= banco_de_dados id . <Classes>
<Classes>         ::= <Classe> <Rclas>
<Rclas>           ::= . <Classes> | ε
<Classe>          ::= classe id ; <Superclasse> <Persistente>
                      <Atributos> ; <Chave> ; <Metodos>
<Superclasse>     ::= superclasse id ; | ε
<Persistente>     ::= persistente ; | ε
<Atributos>       ::= <Atributo> <Ratrs>
<Ratrs>           ::= , <Atributos> | ε
<Atributo>        ::= <Tipo> : id | ε
<Tipo>            ::=  numero ( num <Dec> ) |
                       caracter ( num ) |
                       data |
                       objeto ( id )
<Dec>             ::= , num | ε
<Chave>           ::= Chave : <ListaID>
<ListaID>         ::= id <RLID>
<RLID>            ::= + <ListaID> | ε
<Metodos>         ::= <Metodo> <Rmets>
<Rmets>           ::= ; <Metodos> | ε
<Metodo>          ::= metodo <Tipo_metodo>id
<Tipo_metodo>     ::= atribui_ |
                      le_
```

A manipulação dos dados do protótipo é feita através dos métodos definidos nas classes. É através deles que os atributos dos objetos serão alterados. Foi definida uma linguagem de manipulação de dados (LMD) que mostra como os métodos são executados e que métodos podem ser executados.

O Quadro 11 demonstra a LMD definida para o protótipo, onde as palavras em negrito indicam os métodos disponíveis para execução, com exceção do comando abra que serve para abrir um banco de dados pré-definido através da LDD.

Quadro 11 - Definição da Linguagem de Manipulação de Dados (LMD)

```
Abra <Nome do Banco>;
<Classe>.Lista_Estrutura();
<Objeto> = <Classe>.Cria();
<Objeto> = <Classe>.Posiciona(<Chave>);
<Objeto>.Destroi();
<Objeto>.Persistir();
<Objeto>.Le_<Atributo>();
<Objeto>.Atribui_<Atributo>(<Valor>);
```

Existem dois tipos de métodos disponíveis no protótipo: os métodos de classe e os métodos de instância. A Tabela 1 demonstra os métodos disponíveis no protótipo informando seu tipo e sua aplicação.

Tabela 1 – Métodos disponíveis no protótipo

Método	Tipo	Aplicação
Lista_Estrutura()	Classe	Lista a estrutura da classe do dicionário de dados aberto.
Cria()	Classe	Instancia um objeto vazio da referida classe.
Posiciona()	Classe	Instancia um objeto já existente no banco de dados da referida classe.
Destroi()	Instância	Elimina a instância do objeto em uso.
Persistir()	Instância	Persiste (grava) a instância do objeto em uso.
Le_<Atributo>()	Instância	Lê o conteúdo de um atributo do objeto em uso.
Atribui_<Atributo>()	Instância	Atribui um valor para um atributo do objeto em uso.

4.1 ESPECIFICAÇÃO DO PROTÓTIPO

A especificação do protótipo foi realizada utilizando a linguagem de modelagem *Unified Modeling Language* (UML). A UML é a padronização da linguagem de

desenvolvimento orientado a objetos para visualização, especificação, construção e documentação de sistemas.

Para a modelagem foram utilizados os diagramas de casos de uso, diagramas de seqüência e diagrama de classes. Os mesmos foram feitos utilizando a ferramenta *Rational Rose* da empresa *Rational Software Corp*.

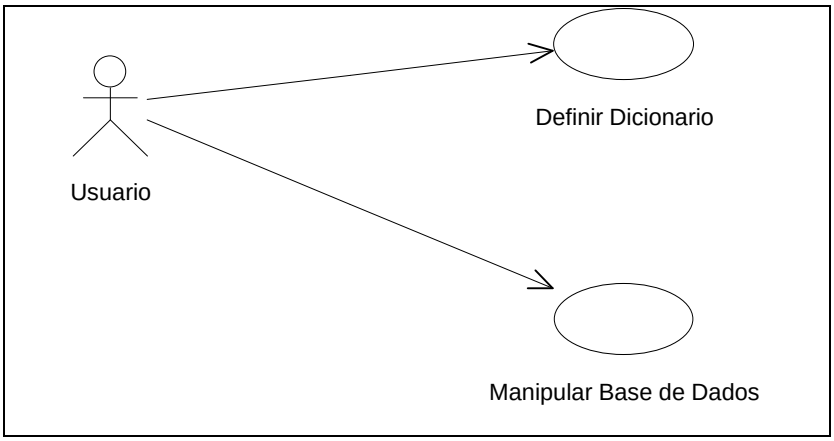
4.1.1 CASOS DE USO

Neste protótipo existem dois casos de uso. Eles são descritos a seguir:

a) definir dicionário: o usuário define as classes com seus métodos e atributos, em um banco de dados através de um *script*. Este *script* é compilado e executado através de uma LDD (Linguagem de Definição de Dados) pré-definida. O dicionário de dados será gravado em um arquivo com extensão DIC, onde serão guardadas todas as definições do banco de dados;
b) manipular base de dados: depois de definido o Dicionário de Dados o usuário manipula a base de dados através dos métodos definidos para as classes do banco de dados. Estes métodos são executados obedecendo a uma LMD (Linguagem de Manipulação de Dados) pré-definida.

A Figura 5 demonstra os dois casos de uso descritos anteriormente.

Figura 5 - Casos de uso do protótipo



4.1.2 DIAGRAMA DE CLASSES

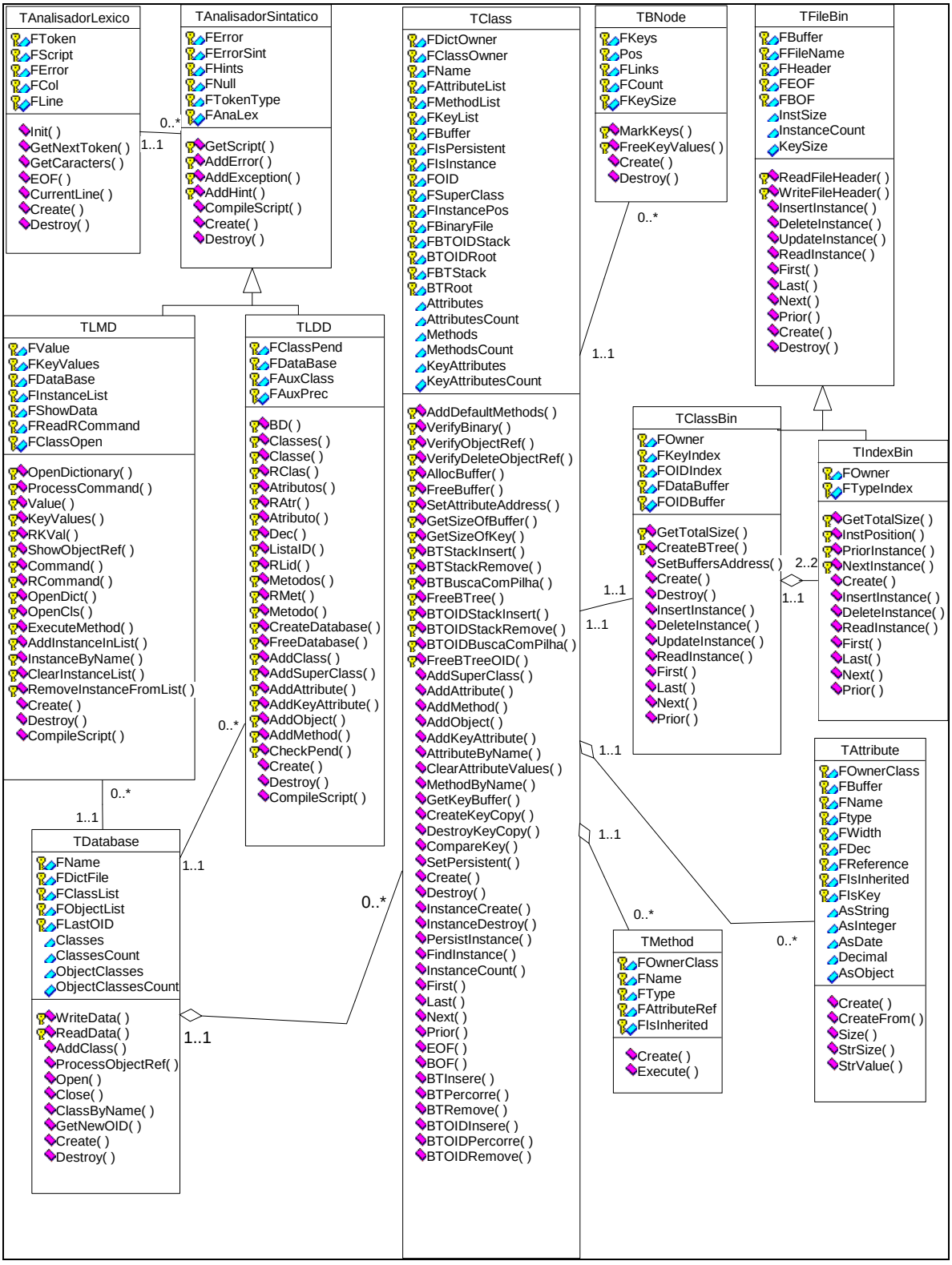
A Figura 6 demonstra o diagrama de classes do protótipo. Os atributos das classes possuem métodos *get* e *set* quando necessário a leitura e atribuição de seus valores por outras classes. Desta maneira mesmo os atributos públicos das classes não podem ser acessados diretamente, elas serão acessadas somente através de propriedades, que é uma construção do Delphi, fazendo com que os atributos continuem encapsulados.

Pode-se verificar a existência das seguintes classes no protótipo:

a) TAnalisadorLexico: responsável pela análise léxica do *script*. A classe faz a identificação do tipo e o valor dos *tokens* do *script*;
b) TAnalisadorSintatico: classe mãe das classes responsáveis pela análise sintática;
c) TLDD: herda as características da classe TAnalisadorSintatico. Implementa a compilação de *scripts* da LDD, isto é só interpreta *scripts* para criação de banco de dados;
d) TLMD: como a TLDD, herda as características de TAnalisadorSintatico, mas interpreta *scripts* LMD, ou seja, funções de abertura de dicionário, listagem da estrutura das classes e execução dos métodos;
e) TDatabase: nesta classe estão armazenadas as lista das classes (TClass) do dicionário. Possui métodos que adicionam classes na respectiva lista. Para tornar as definições do dicionário de dados persistentes, existem os métodos *ReadData* e *WriteData* que lêem e escrevem em um arquivo binário (arquivo com extensão DIC) as informações do dicionário de dados, como a lista de classes, a lista de ligações entre as classes, a lista de atributos (TAttribute) e a lista de métodos (TMethod) de cada classe;
f) TClass: nesta classe estão armazenadas as listas de atributos e métodos. Possui métodos para adicionar e procurar atributos e métodos nas respectivas listas. Possui também métodos para adicionar campos na chave da classe. A classe possui um *buffer* de memória que contém os dados dos atributos, o tamanho deste *buffer* é o mesmo que a soma de todos os valores da propriedade “Size” dos objetos TAttribute na lista. Quando um atributo é incluído na lista de atributos, o endereço do *buffer* mais o deslocamento em bytes dentro do *buffer* é atribuído ao ponteiro contido no objeto TAttribute;

g) TAttribute: nesta classe são definidas as propriedades de cada atributo de uma classe do banco de dados;

Figura 6 – Diagrama de classes do protótipo



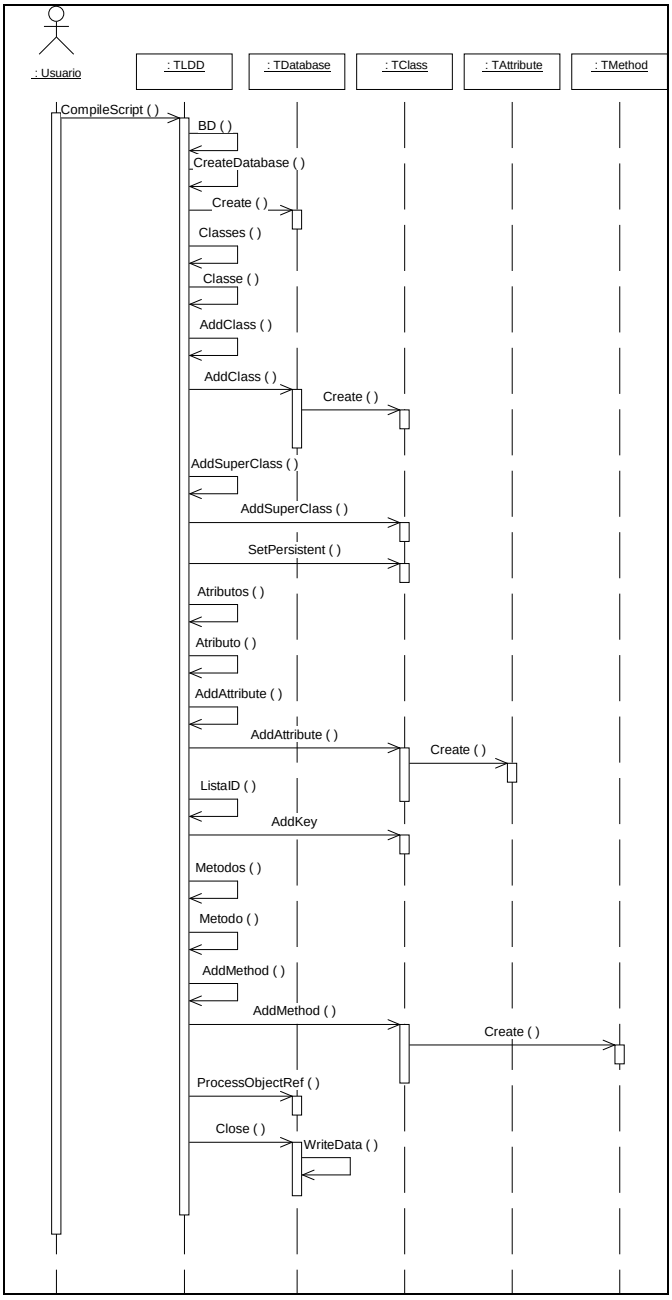
h) TMethod: nesta classe são definidas as propriedades de cada método de uma classe do banco de dados. É somente através dos métodos que é possível se fazer a manipulação dos dados no banco de dados;
i) TFileBin: Classe mãe das classes TIndexBin e TClassBin que tem a finalidade de fazer as operações com os arquivos binários;
j) TClassBin: nesta classe é implementada a manipulação de dados no arquivo DAT;
k) TIndexBin: nesta classe é implementada a indexação do arquivo DAT, que gera um arquivo IKEY, que é um índice pela chave da classe e outro arquivo IOID, que é um índice pelo OID. Estes arquivos funcionam como uma lista duplamente encadeada;
l) TBNode: esta é a classe base para implementação da árvore B.

4.1.3 DIAGRAMAS DE SEQUÊNCIA

Os diagramas de seqüências representam a seqüência em que as ações ocorrem dentro do sistema. Eles demonstram como é feita a troca de mensagens entre as classes. Existe um diagrama de seqüência para o caso de uso “Definir Dicionário”, e para o caso de uso “Manipular Base de Dados” existe um diagrama principal que mostra a execução do processo até a execução do método. Depois foram feitos diagramas específicos para cada método.

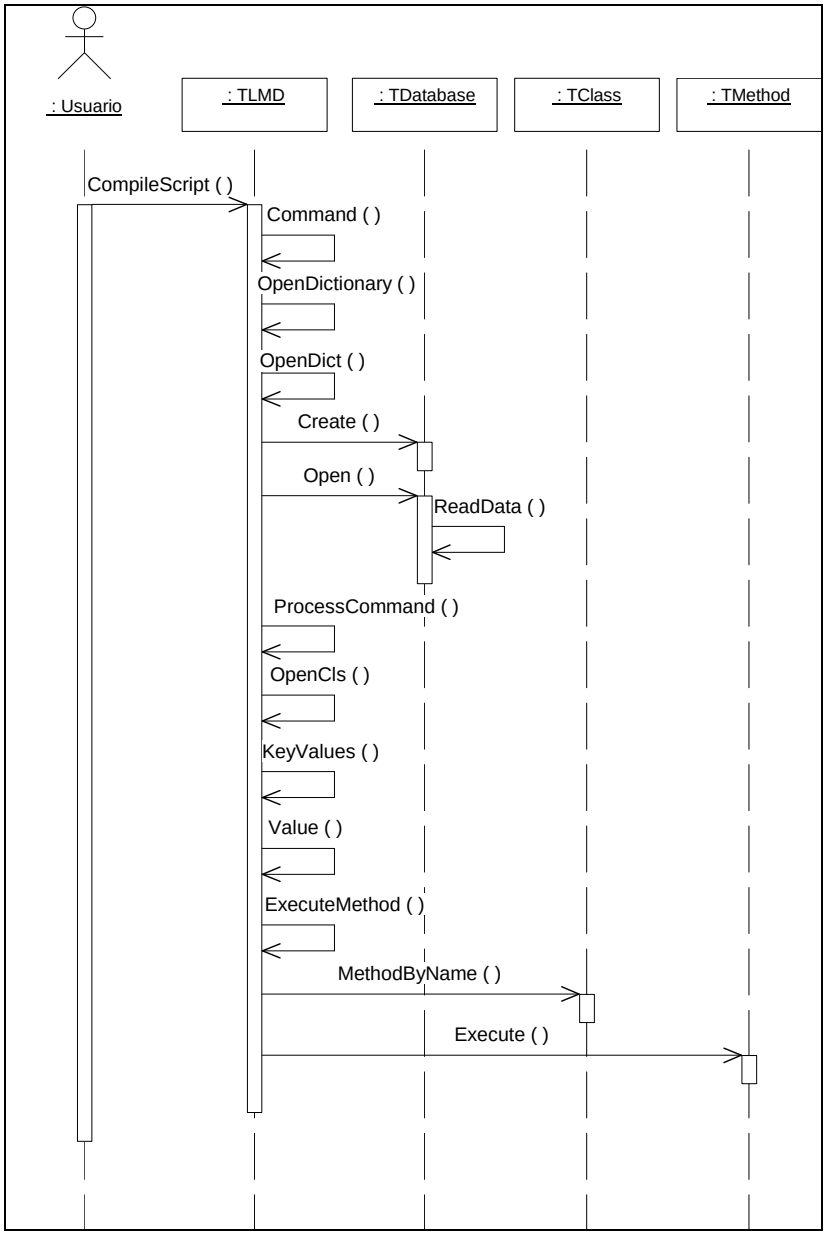
A Figura 7 mostra o processo de execução de um script de definição de dados, que irá gerar o dicionário do banco de dados.

Figura 7 – Diagrama de seqüência da execução de uma LDD



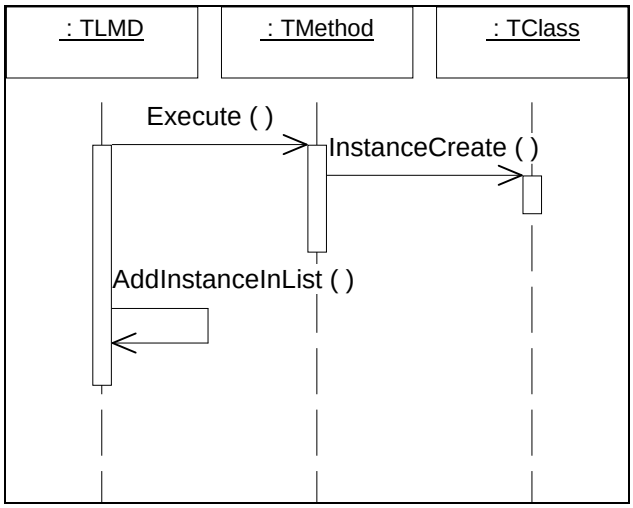
A Figura 8 mostra o processo de execução de um *script* de manipulação de dados, que irá executar um método de uma classe.

Figura 8 – Execução de um *script* de manipulação de dados



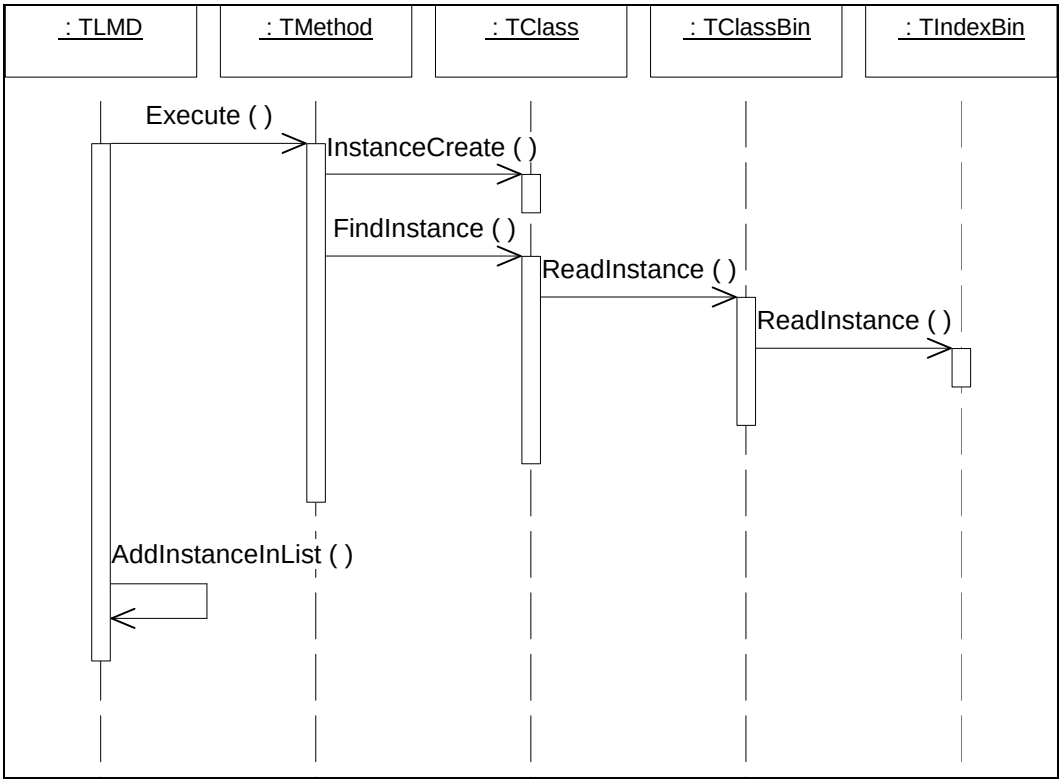
A Figura 9 demonstra a continuação do processo de execução de um método para o método Cria.

Figura 9 – Execução do método Cria



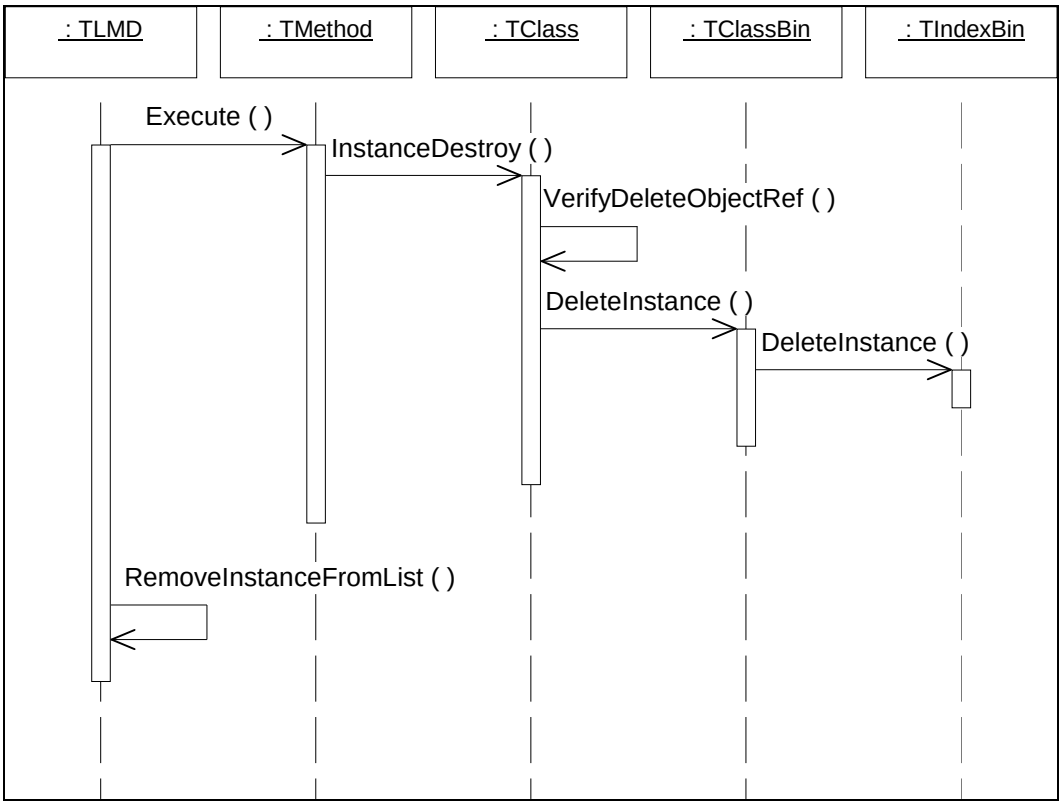
A Figura 10 demonstra a continuação do processo de execução de um método para o método Posiciona.

Figura 10 – Execução do método Posiciona



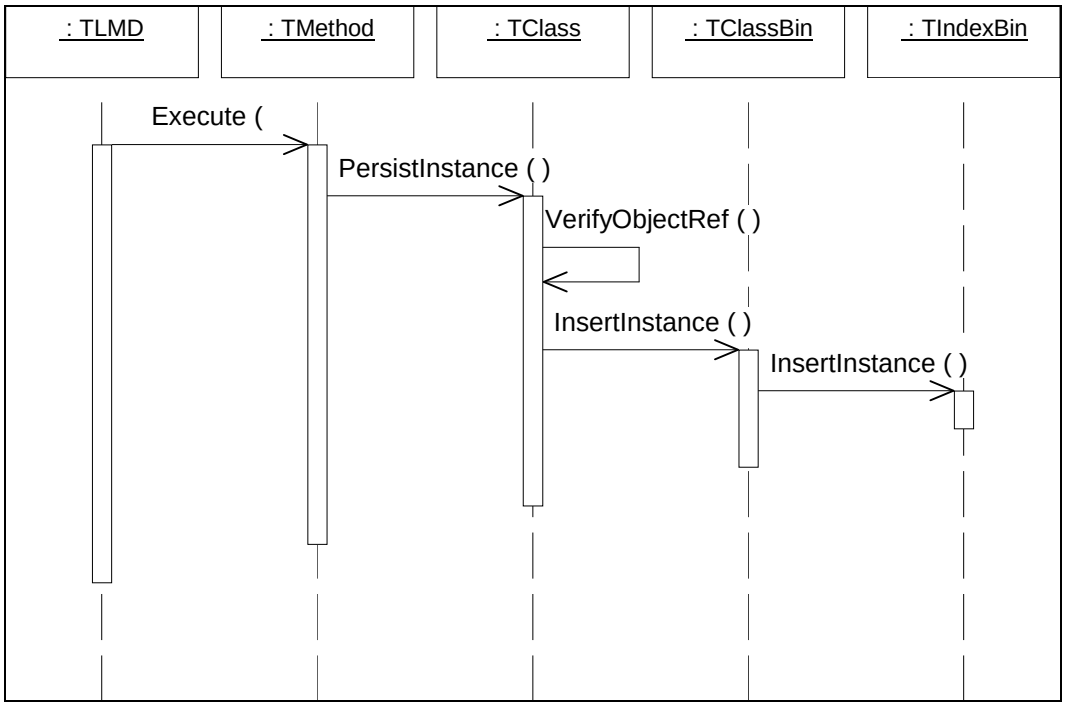
A Figura 11 demonstra a continuação do processo de execução de um método para o método Destroi.

Figura 11 – Execução do método Destroi



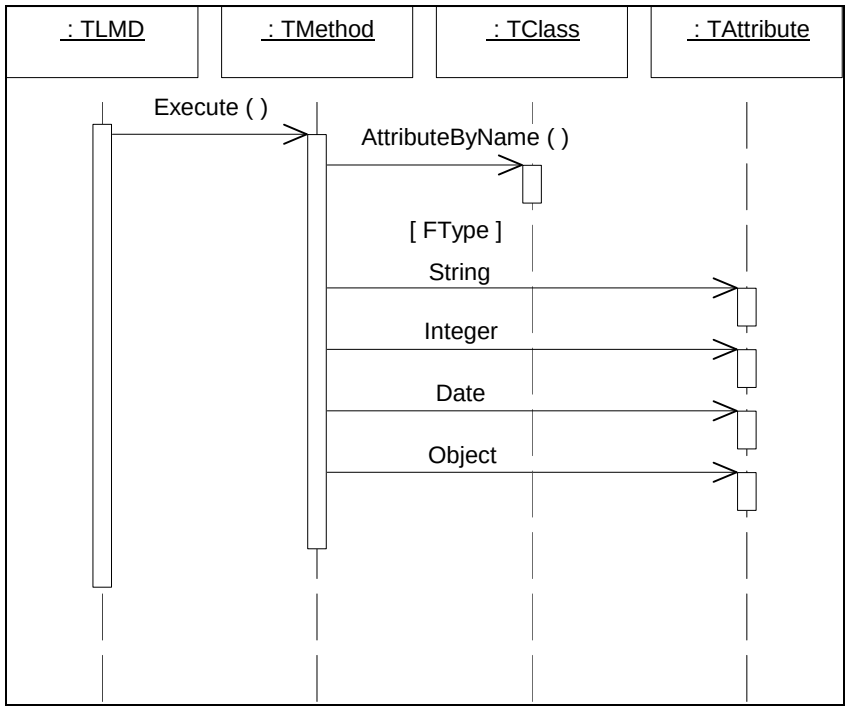
A Figura 12 demonstra a continuação do processo de execução de um métodos para o método Persistir.

Figura 12 - Execução do método Persistir



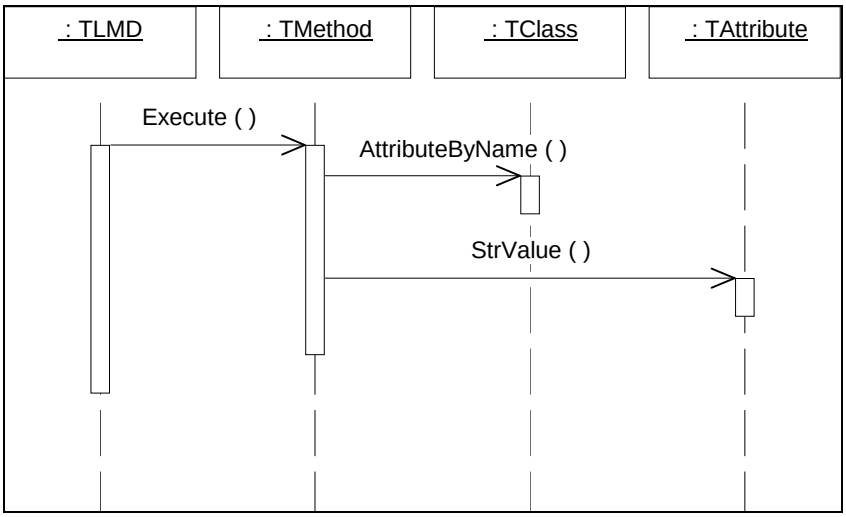
A Figura 13 demonstra a continuação do processo de execução de um métodos para o método Atribui.

Figura 13 - Execução do método Atribui



A Figura 14 demonstra a continuação do processo de execução de um métodos para o método Le.

Figura 14 – Execução do método Le



4.2 IMPLEMENTAÇÃO

Para implementação do protótipo foi utilizada a ferramenta Borland Delphi 5.0. O Delphi é um ambiente integrado de desenvolvimento que permite a edição compilação e execução de programas. A linguagem de programação utilizada pelo Delphi é o Object Pascal.

Para a manipulação física das instâncias das classes foi criado um sistema onde existem: um arquivo de dados (com extensão DAT) que não tem nenhuma organização, dois arquivos seqüenciais indexados e duas árvores B. Existe um tipo de arquivo para cada classe definida no dicionário de dados.

Um dos arquivos seqüenciais indexados contém as chaves das instâncias, este arquivo possui a extensão IKEY. O outro arquivo contém os OIDs das instâncias e possui a extensão IOID. Ambos os arquivos guardam também a posição da instância no arquivo DAT. Já as árvores B são geradas a partir dos arquivos de chaves e de OIDs e apontam para as posições físicas destes arquivos.

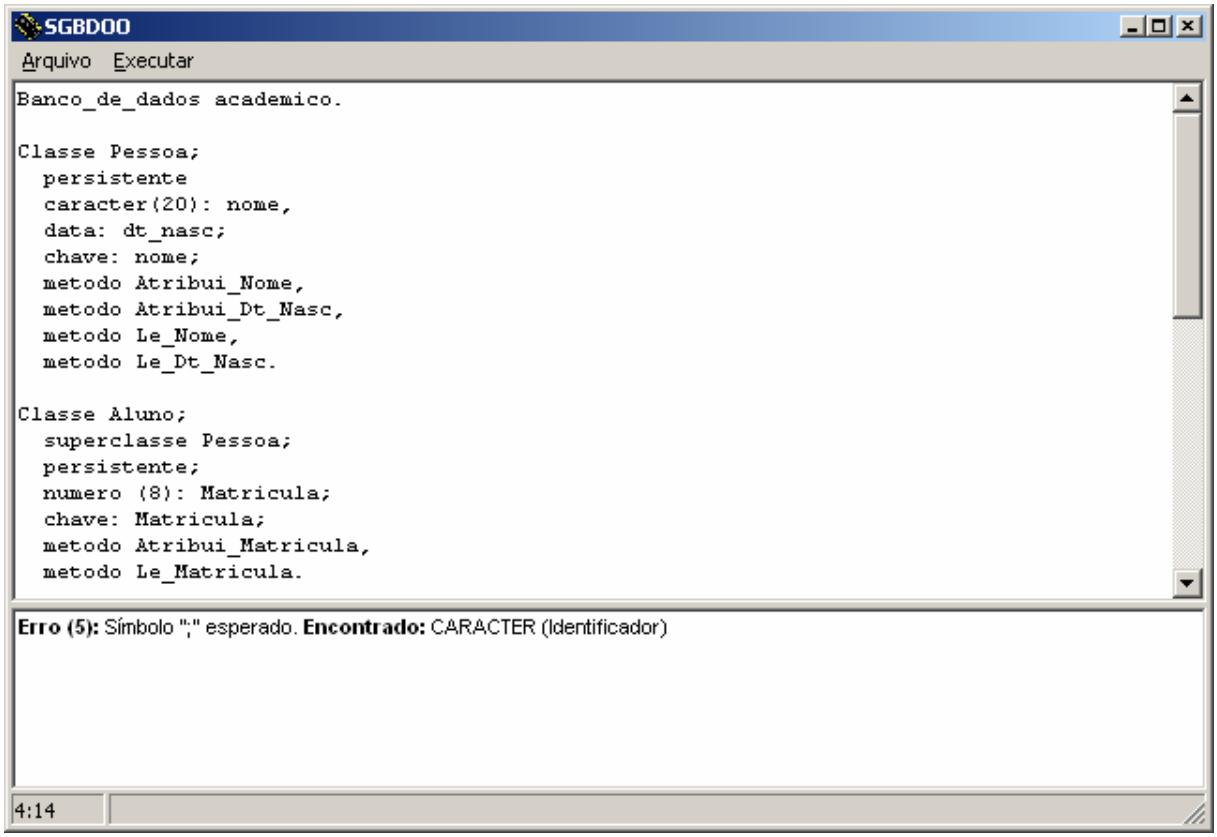
Desta maneira quando é necessário fazer uma procura por uma instância de uma classe, esta procura é feita em uma das árvores de acordo com o dado utilizado para a procura

(Chave ou OID). Uma vez encontrada a posição da instância dentro do arquivo de dados é feita a leitura dos dados da mesma.

Existe no protótipo a opção de salvar e abrir um *script*. Estes *scripts* são salvos em arquivos com a extensão SBD e tem o formato texto, sendo que é permitido abrir qualquer arquivo texto.

A Figura 15 mostra a tela principal do protótipo. Nela encontra-se o editor onde são digitados os *scripts* de definição e manipulação dos dados e um quadro onde são mostrados os erros encontrados na compilação de um *script*. No menu Arquivo encontram-se os itens Abrir e Salvar (para os *scripts* definidos no editor), a chamada para a tela de opções e o item sair. No menu Executar encontra-se os item Executar LDD, Executar LMD e Limpar.

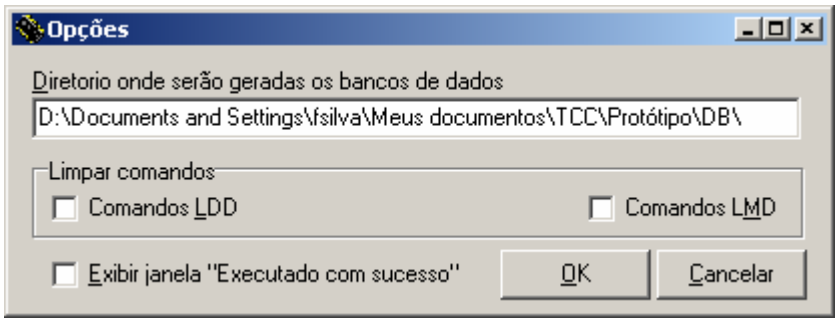
Figura 15 – Tela principal do protótipo



A Figura 16 mostra a tela de opção do protótipo, onde é informado o diretório onde serão gerados os bancos de dados. Também nesta tela define-se se após a execução dos

comandos LDD e LMD deve ser limpa a tela e se deve ser exibida uma tela informando que o *script* foi executado com sucesso.

Figura 16 – Tela de opções



4.2.1 EXECUÇÃO DE UM ESTUDO DE CASO

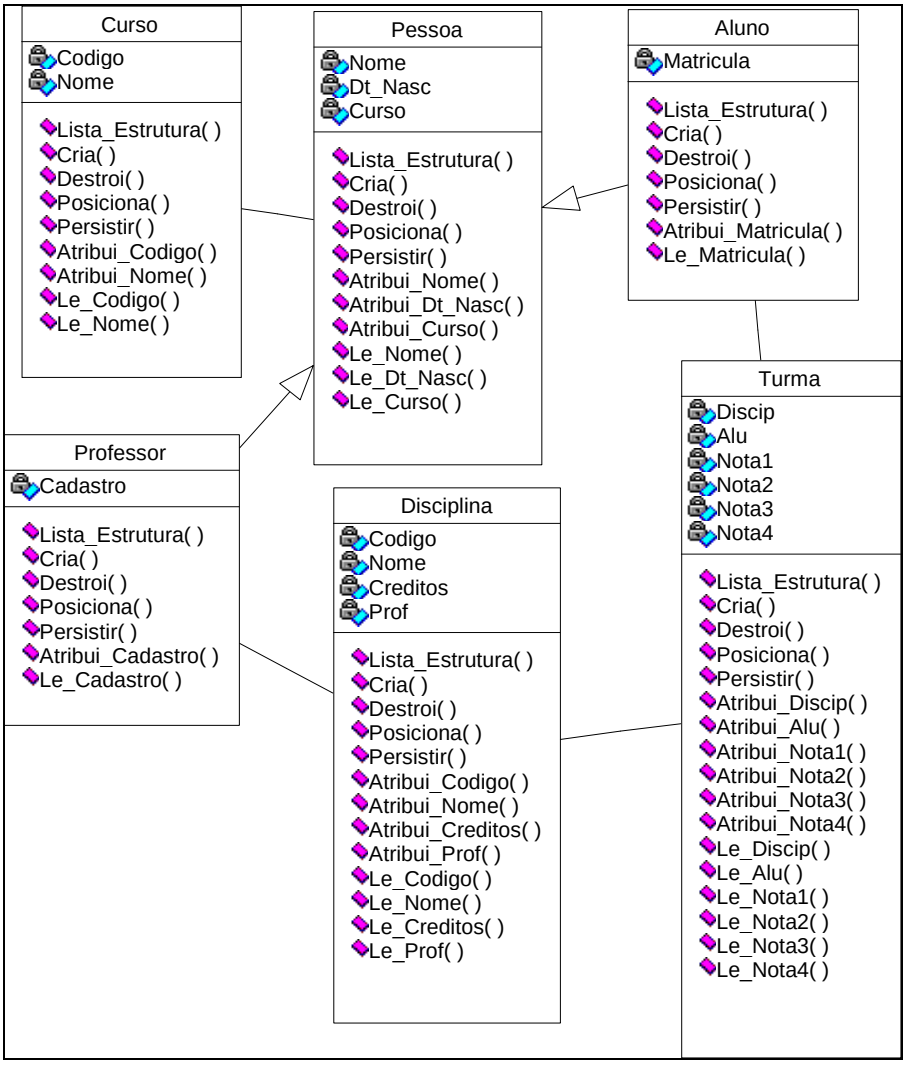
Para um melhor entendimento do funcionamento do protótipo e para demonstrar os conceitos de OO que são implementados no mesmo, será feita a execução de um estudo de caso.

A descrição do estudo é a seguinte: deseja-se definir um dicionário de dados com as seguintes classes persistentes:

a) Curso: esta classe possui os atributos Código e Nome. Sua chave é o atributo código;
b) Pessoa: esta classe possui os atributos Nome, Dt_Nasc e uma referência para a classe Curso. A chave da classe é o atributo Nome;
c) Aluno: classe derivada da classe Pessoa. Possui o atributo Matricula que é a sua chave;
d) Professor: esta classe também é derivada da classe Pessoa. Possui o atributo Cadastro que é a sua chave;
e) Disciplina: possui os atributos Codigo, Nome, Creditos e uma referência para a classe Professor, sua chave é o atributo Codigo;
f) Turma: é a ligação entre uma disciplina e seus alunos, possui uma referência para classe Disciplina e outra para a classe Aluno, os quais formam a chave da classe. Possui também os atributos Nota1, Nota2, Nota3 e Nota4 que indicam as notas do aluno na respectiva disciplina.

A Figura 17 mostra o diagrama de classes do estudo de caso descrito acima, onde os métodos Cria, Destroi, Persistir, Posiciona e Lista_Estrutura são métodos gerados automaticamente pelo sistema.

Figura 17 – Diagrama de classes do estudo de caso



Todos as classes possuem métodos para atribuir e ler os valores dos atributos das mesmas. Após a definição do dicionário de dados, será feita a manipulação destes dados, por exemplo, incluir novas instâncias e alterar seus atributos.

O Quadro 12 mostra o *script* de definição dos dados que será executado para definir o dicionário de dados. Esta execução irá gerar um arquivo com a extensão DIC, o nome do arquivo é o mesmo informado na primeira linha do *script* precedido do identificador “banco_de_dados”.

Quadro 12 – *Script* de geração do dicionário de dados

```
Banco_de_dados Exemplo.

Classe Curso;
  persistente;
  numero(6): Codigo,
  caracter(20): Nome;
  chave: Codigo;
  metodo Atribui_Codigo,
  metodo Atribui_Nome,
  metodo Le_Codigo,
  metodo Le_Nome.

Classe Pessoa;
  persistente;
  caracter(20): Nome,
  data: Dt_Nasc,
  objeto(Curso): Curso;
  chave: nome;
  metodo Atribui_Nome,
  metodo Atribui_Dt_Nasc,
  metodo Atribui_Curso,
  metodo Le_Nome,
  metodo Le_Dt_Nasc,
  metodo Le_Curso.

Classe Aluno;
  superclasse Pessoa;
  persistente;
  numero (8): Matricula;
  chave: Matricula;
  metodo Atribui_Matricula,
  metodo Le_Matricula.

Classe Professor;
  superclasse Pessoa;
  persistente;
  numero (8): Cadastro;
  chave: Cadastro;
  metodo Atribui_Cadastro,
  metodo Le_Cadastro.

Classe Disciplina;
  persistente;
  numero(6): Codigo,
  caracter(20): Nome,
  numero(2): Creditos,
  objeto(Professor): Prof;
  chave: Codigo;
  metodo Atribui_Codigo,
  metodo Atribui_Nome,
  metodo Atribui_Creditos,
  metodo Atribui_Prof,
  metodo Le_Codigo,
  metodo Le_Nome,
  metodo Le_Creditos,
  metodo Le_Prof.

Classe Turma;
  persistente;
  objeto(Disciplina): Discip,
  objeto(Aluno): Alu,
  numero(2,2): Nota1,
  numero(2,2): Nota2,
  numero(2,2): Nota3,
  numero(2,2): Nota4;
  chave: Discip + Alu;
  metodo Atribui_Discip,
  metodo Atribui_Alu,
  metodo Atribui_Nota1,
  metodo Atribui_Nota2,
```

```
        metodo Atribui_Nota3,
        metodo Atribui_Nota4,
        metodo Le_Discip,
        metodo Le_Alu,
        metodo Le_Nota1,
        metodo Le_Nota2,
        metodo Le_Nota3,
        metodo Le_Nota4.
```

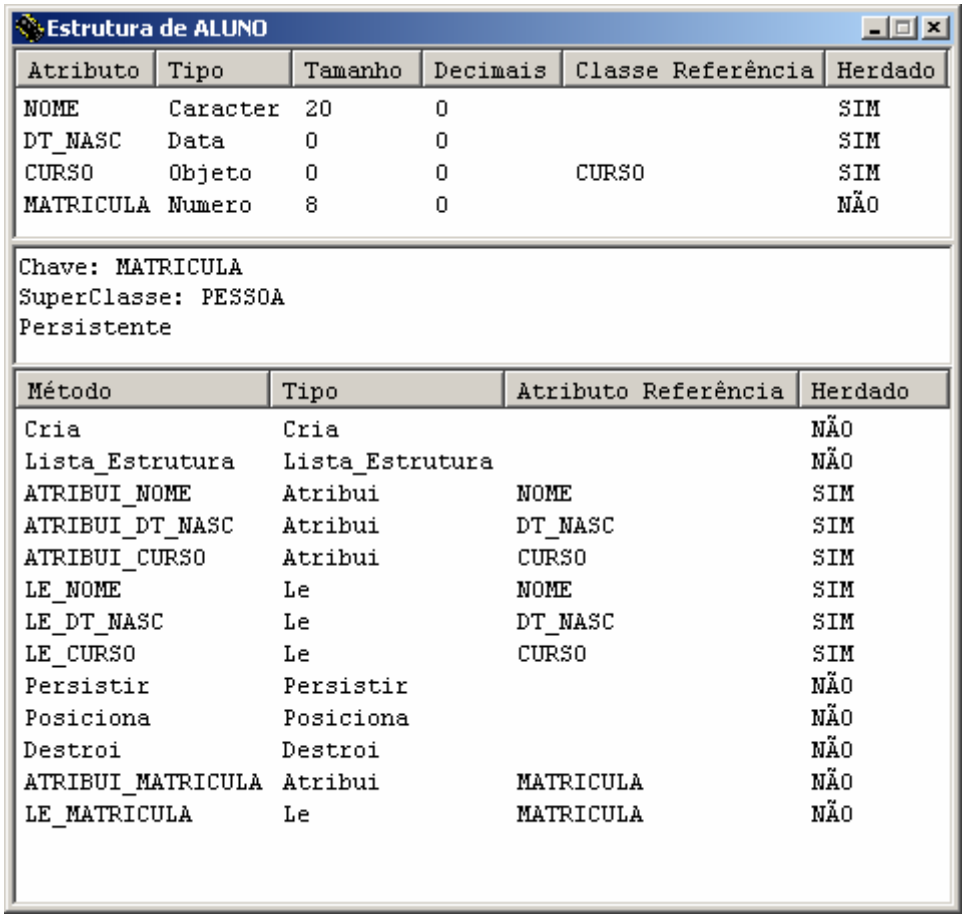
O que determina se uma classe é persistente é a cláusula “Persistente” na sua definição. A existência desta cláusula faz com que o sistema disponibilize automaticamente para a classe os métodos Persistir, Posiciona e Destroi. Existem dois métodos que são disponibilizados para todas as classes que são os métodos Cria e Lista_Estrutura.

Já na definição do dicionário de dados pode-se verificar alguns dos conceitos de OO no SGBDOO, como a utilização de classes. O conceito de herança pode ser verificado na definição da cláusula “Superclasse <Classe>” onde a classe que está sendo definida irá herdar os atributos e métodos da classe informada.

Outro conceito é o de tipos abstratos de dados que podem ser verificados nos atributos Discip e Alu que são atributos dos tipos Disciplina e Aluno respectivamente.

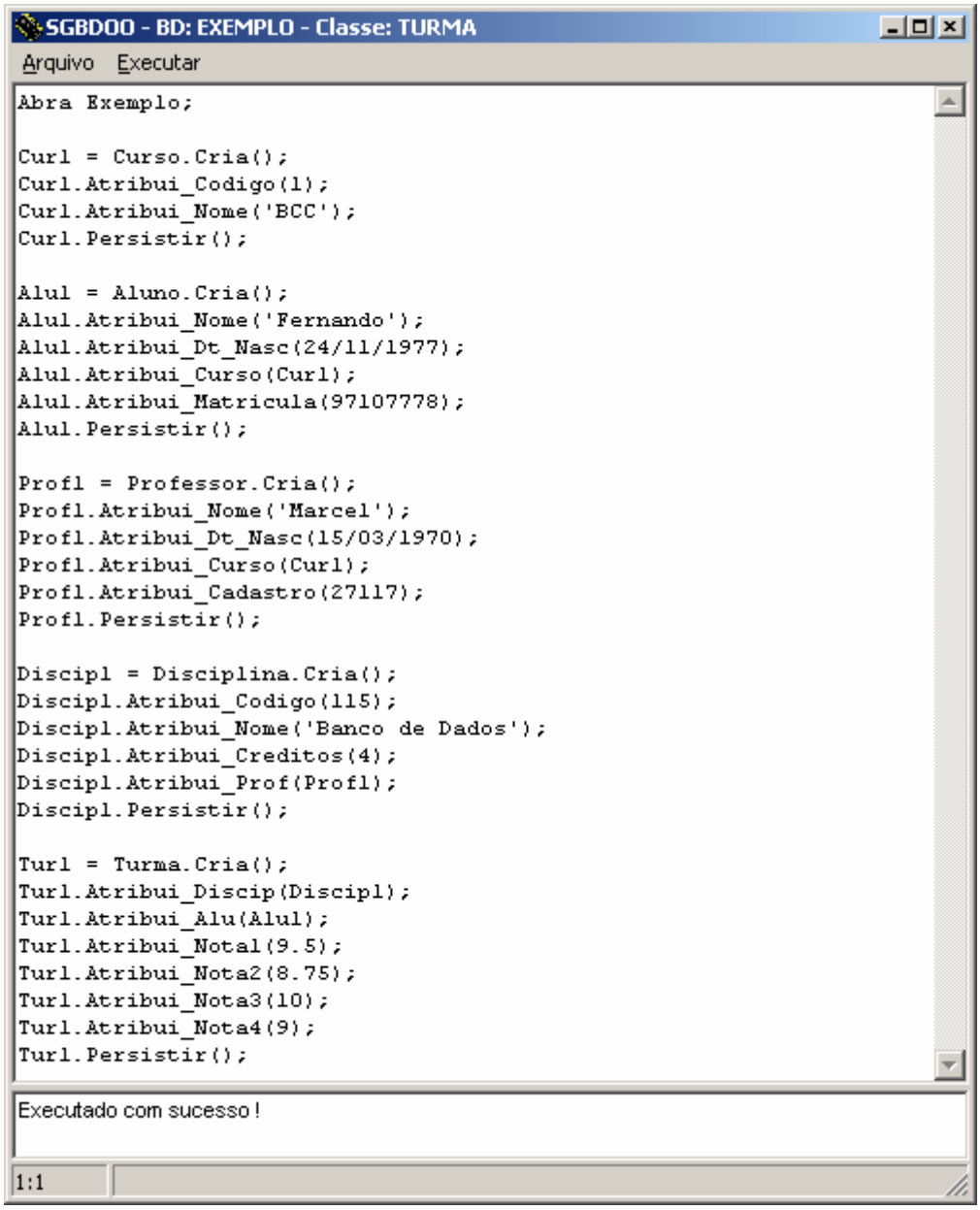
A Figura 18 mostra o resultado da execução do método Lista_Estrutura, que mostra a estrutura da classe Aluno. Pode-se notar que na estrutura estão sendo mostrados também os atributos e métodos herdados da superclasse Pessoa.

Figura 18 – Execução do método Lista_Estrutura da classe Aluno



A Figura 19 mostra um exemplo de manipulação de dados, onde é feito a criação, atribuição de valores e persistência de instâncias das classes Aluno, Professor e Disciplina.

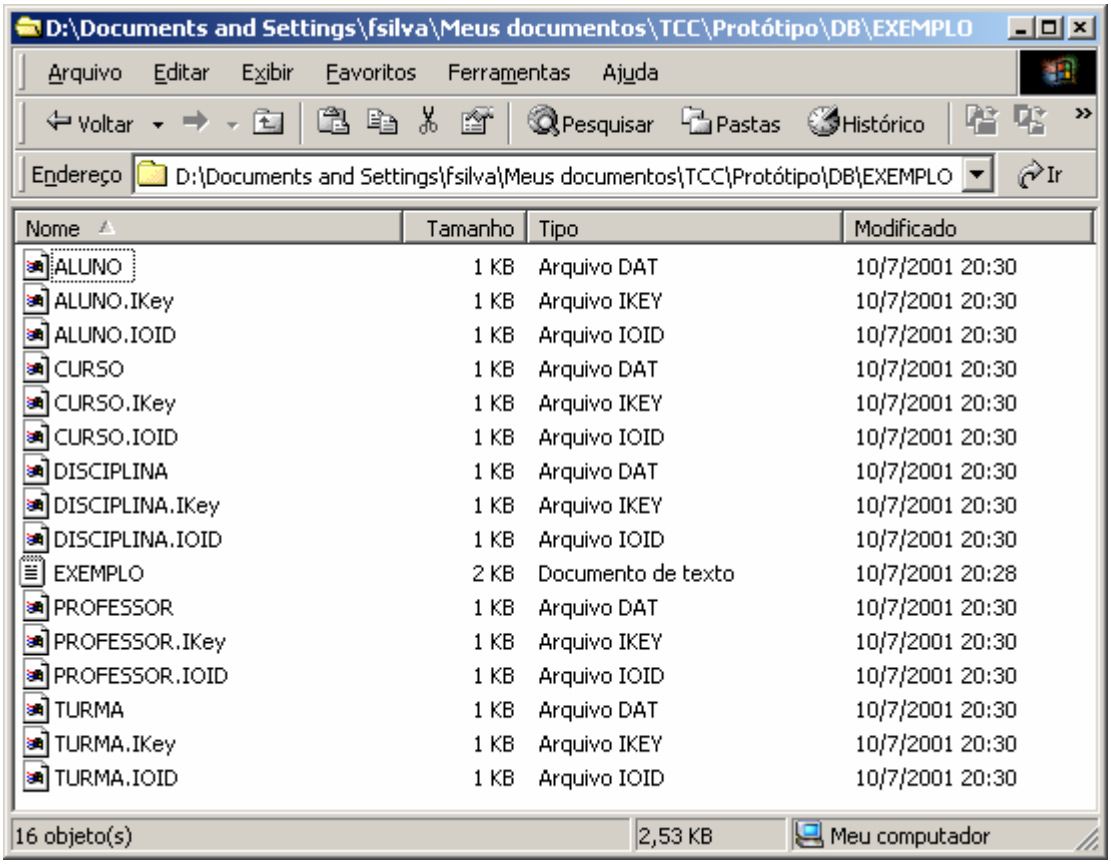
Figura 19 – Exemplo criação de instâncias no protótipo



No momento da criação de uma nova instância de uma classe o sistema gera automaticamente o OID para a instância. Este OID é único e não muda durante toda a existência do objeto. A liberação dos objetos declarados durante a execução dos *scripts* é feita pelo próprio sistema que se encarrega disto quando um banco de dados é fechado. Isto ocorre quando um novo banco é aberto ou quando o sistema é finalizado. Caso uma variável de instância que já exista receba uma nova atribuição o sistema libera a instância anterior e faz a atribuição da nova.

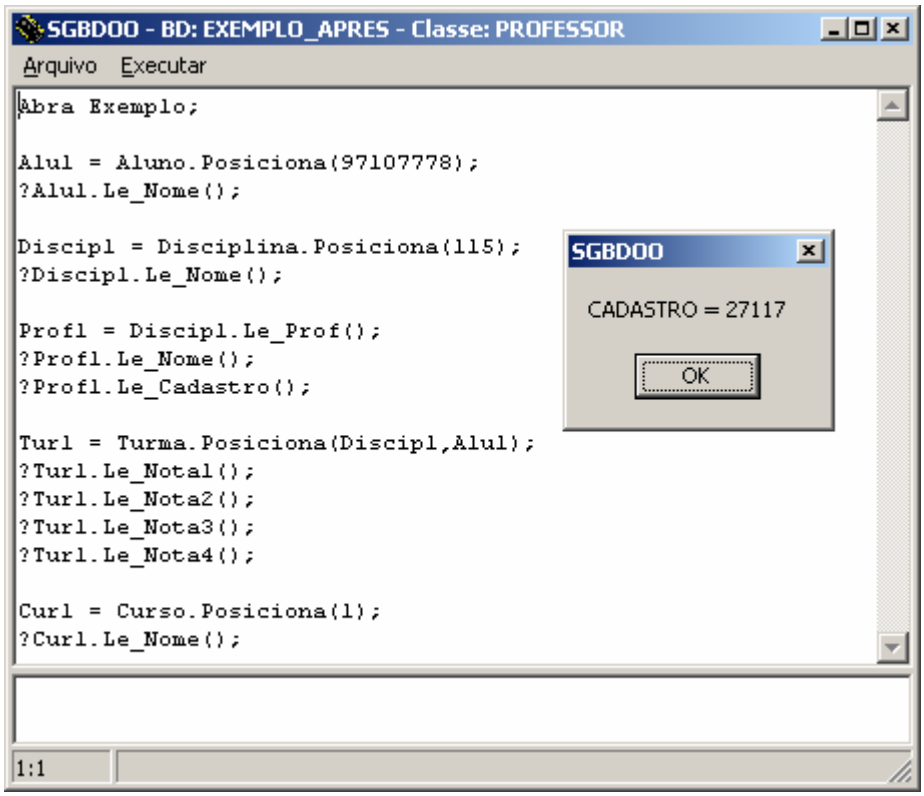
A Figura 20 mostra os arquivos gerados após a execução do *script* criação das instâncias, são gerados três arquivos para cada classe, um com os dados (.DAT) e dois com os índices (.IKEY e .IOID).

Figura 20 – Arquivos gerados pelo script de criação das instâncias



A Figura 21 mostra um *script* onde é feito a leitura de instâncias já existentes no banco de dados. Mostra também a leitura dos atributos destas classes através de seus métodos. O caracter “?” na frente dos comandos de leitura faz com que o valor do atributo seja mostrado na tela, a janela menor da figura mostra o valor do atributo Cadastro da instância Prof1 carregada a partir da instância Discip1.

Figura 21 – Exemplo de *script* para leitura de instâncias do banco de dados



5 CONCLUSÕES

Este capítulo irá apresentar as conclusões com relação ao trabalho desenvolvido e sugestões para trabalhos futuros.

Com o crescimento da utilização dos conceitos OO para desenvolvimento de aplicações de alto nível, torna-se cada vez mais necessário que se utilize uma forma de armazenamento em que os objetos possam ser modelados e armazenados sem que haja a necessidade de transformação dos mesmos.

As necessidades de funcionalidade e performance da nova dimensão da computação exigem bancos de dados que transcendam os limites do modelo relacional. Com a utilização dos conceitos de OO é possível uma representação realista do mundo dos dados complexos.

A união das aptidões de bancos de dados e os conceitos OO tornam os SGBDOO poderosas ferramentas dentro do processo de desenvolvimento de software, visto que unem a produtividade do desenvolvimento orientado a objetos com a segurança dos SGBD.

O trabalho alcançou o seu objetivo, visto que foi possível demonstrar conceitos como classes, herança, tipos de dados abstratos e identidade de objeto, na implementação do protótipo. Mostrando assim a facilidade em se modelar e manipular modelos orientados a objetos dentro de um SGBDOO. Das linguagens de definição de dados dos bancos de dados estudados a que mais se assemelha a LDD do protótipo é a Caché Class Description Language (CDL) do banco de dados Caché.

O protótipo tem grande aplicação didática sendo possível utilizá-lo em disciplinas de banco de dados para exemplificar a aplicação dos conceitos de OO dentro de um SGBDOO. A flexibilidade do mesmo possibilita que sejam incorporados no futuro outros conceitos de OO e até as aptidões de um Banco de dados com certa facilidade.

Um conceito de OO que não foi demonstrado no protótipo, mas não menos importante é o conceito de agregação por valor ou composição, que é quando uma classe agrega ou é formada por outras classes (partes), formando um todo.

O trabalho contribui muito para o entendimento do funcionamento dos bancos de dados orientados a objetos, esclarecendo dúvidas e fixando novos conceitos sobre esta tecnologia.

Os SGBDOO embora sejam caracterizados como sistemas que satisfazem as necessidades de aplicação avançadas, podem beneficiar todos os tipos de aplicação de banco de dados. Mesmo assim deve-se escolher o tipo de banco de dados de acordo com o tipo dos dados e a maneira que eles serão acessados pelos usuários, pois não existe um SGBD compatível com aplicações diversas.

5.1 SUGESTÕES

Como sugestão para trabalhos futuros estão relacionados alguns itens que poderiam ser incorporados ao protótipo, a fim de complementá-lo:

a) implementar o conceito de agregação por valor ou composição no protótipo, que é uma característica de OO que não foi implementada;
b) implementar uma linguagem para definição dos métodos, de maneira que seja possível ao usuário programar os seus métodos e não somente utilizar os disponibilizados pelo sistema;
c) incorporar ao protótipo também as aptidões de um banco de dados como transações, concorrência, recuperação, versionamento e restrições de integridade;
d) permitir o compartilhamento dos dados e acesso dos mesmos via rede;
e) criar interface com linguagens de programação;
f) desenvolver alguma aplicação de cunho prática, utilizando o protótipo.

ANEXO I

Este anexo apresenta as interfaces das classes do protótipo.

a) Interface da classe TAnalisadorLexico:

```
unit uLexico;

interface

uses Classes, SysUtils;

type
  TTokenType = (ttId, ttNum, ttSymbol, ttUnknow);

  TAnalisadorLexico = class
  protected
    FError,
    FScript: TStringList;
    FLine,
    FCol: Cardinal;
    FToken: String;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Init;
    function GetNextToken: TTokenType;
    function GetCaracters(Ch: Char): String;
    function EOF: Boolean;
    function CurrentLine: Integer;
    property Token: String read FToken;
    property Script: TStringList read FScript;
    property Error: TStringList read FError;
  end;
```

b) Interface das classes TanalisadorSintatico, TLDD e TLMD:

```
unit uSintatico;

interface

uses Dialogs, Classes, SysUtils, uLexico, uDict;

type
  TAnalisadorSintatico = class
  protected
    FError,
    FErrorSint,
    FHints: TStringList;
    FNull: Boolean;
    FTokenType: TTokenType;
    FAnaLex: TAnalisadorLexico;

    function GetScript: TStringList;
    function GetError: TStringList;
    procedure AddError(const FmtStr, Erro: String);
    procedure AddException(const ExceptMsg: String);
    procedure AddHint(const Hint: String);
  public
    constructor Create; virtual;
    destructor Destroy; override;
    procedure CompileScript; virtual;
    property Script: TStringList read GetScript;
    property Error: TStringList read GetError;
    property Hints: TStringList read FHints;
  end;

  TLDD = class(TAnalisadorSintatico)
  protected
    FClassPend: TStringList;
    FDataBase: TDataBase;
    FAuxClass: TClass;
```

```
  FAuxPrec: Byte;
  // métodos sintáticas (Sin)
  function BD: Boolean;
  function Classes: Boolean;
  function Classe: Boolean;
  function RClas: Boolean;
  function Atributos: Boolean;
  function RAtr: Boolean;
  function Atributo: Boolean;
  function Dec: Boolean;
  function ListaID: Boolean;
  function RLid: Boolean;
  function Metodos: Boolean;
  function RMet: Boolean;
  function Metodo: Boolean;
  // métodos semânticas (Sem)
  procedure CreateDataBase(DatabaseName: String);
  procedure FreeDataBase;
  procedure AddClass(ClassName: String);
  procedure AddSuperClass(SuperClassName: String);
  procedure AddAttribute(AttributeName: String; AttributeType: TAttributeType;
                         Width, Prec: Byte);
  procedure AddKeyAttribute(AttributeName: String);
  procedure AddObject(AttributeName, ReferenceClass: String);
  procedure AddMethod(MethodName, AttributeName: String; MethodType: TMethodType);
  procedure CheckPend;
public
  constructor Create; override;
  destructor Destroy; override;
  procedure CompileScript; override;
  property DataBase: TDataBase read FDataBase;
end;

TInstance = record
  Name: String;
  Instance: TClass;
end;

PInstance = ^TInstance;

TLMD = class(TAnalisadorSintatico)
protected
  FValue: String;
  FKeyValues: TStringList;
  FDataBase: TDataBase;
  FInstanceList: TList;
  FShowData,
  FReadRCommand: Boolean;
  FClassOpen: TClass;
  // métodos sintáticas (Sin)
  function OpenDictionay: Boolean;
  function ProcessCommand(CanSet: Boolean = True): Boolean;
  function Value: Boolean;
  function KeyValues: Boolean;
  function RKVal: Boolean;
  function ShowObjectRef: Boolean;
  // métodos semânticos (Sem)
  function Command: Boolean;
  function RCommand: Boolean;
  procedure OpenDict(const DictFileName: string);
  procedure OpenCls(const ClassName: String);
  procedure ExecuteMethod(const MethodName: String; InstName: String = '';
                          Setting: Boolean = False);
  procedure AddInstanceInList(InstName: String; Instance: TClass);
  function InstanceByName(InstName: String): TClass;
  procedure ClearInstanceList;
  procedure RemoveInstanceFromList(InstName: String);
public
  constructor Create; override;
  destructor Destroy; override;
  procedure CompileScript; override;
  property DataBase: TDataBase read FDataBase write FDataBase;
  property ClassOpen: TClass read FClassOpen write FClassOpen;
end;
```

c) Interface das classes TDatabase, TClass, TAttribute, TMethod, TFileBin, TClassBin, TIndexBin e TBNode:

```
unit uDict;

interface

uses Classes, SysUtils, Registry, Windows;

const
  BTreeNodeKeys = 5;
  HeaderSize = 64;

type
  TAttributeType = (taChar, taNum, taDate, taObject);
  TMethodType = (mtAtribui, mtLe, mtCria, mtDestroi, mtLista_Estrutura, mtPersistir,
                 mtPosiciona);
  TCompOperator = (coEqual, coDifferent, coGreater, coGreaterEqual, coLesser, coLesserEqual);
  TOID = Cardinal;
  TTypeIndex = (tiOID, tiKey);

  TDataBase = class;
  TClass = class;
  TAttribute = class;
  TMethod = Class;
  TClassBin = class;

  // Classes Diversas
  TBNode = class
  protected
    FKeys: array [1..2*BTreeNodeKeys] of Pointer;
    Pos: array [1..2*BTreeNodeKeys] of Longint;
    FLinks: array [0..2*BTreeNodeKeys] of TBNode;
    FCount,
    FKeySize: Integer;
    procedure MarkKeys;
    procedure FreeKeyValues;
  public
    constructor Create(KeySize: Integer);
    destructor Destroy; override;
  end;

  TBTrabNode = class
  protected
    PKeys,
    PPos,
    PLinks: Pointer;
    FCount,
    FKeySize: Integer;

    procedure SetFKeys(Index: Integer; Value: Pointer);
    function GetFKeys(Index: Integer): Pointer;
    procedure SetFLinks(Index: Integer; Value: TBNode);
    function GetFLinks(Index: Integer): TBNode;
    procedure SetPos(Index: Integer; Value: Longint);
    function GetPos(Index: Integer): Longint;
  public
    constructor Create(KeySize, NumEntradas: Integer);
    destructor Destroy; override;
    property FKeys[Index: Integer]: Pointer read GetFKeys write SetFKeys;
    property Pos[Index: Integer]: Longint read GetPos write SetPos;
    property FLinks[Index: Integer]: TBNode read GetFLinks write SetFLinks;
  end;

  TObjectRefType = record
            ForClass,
            RefClass: String;
            end;
  PObjectRefType = ^TObjectRefType;

  // Classes de Dados (Database, Class, Attribute)
  TDataBase = class
```

```
protected
  FName,
  FDictFile: String;
  FClassList,
  FObjectList: TList;
  FLastOID: TOID;
  function GetClass(Index: Integer): TClass;
  function GetClassesCount: Integer;
  function GetObjectClasses(Index: Integer): TObjectRefType;
  function GetObjectClassesCount: Integer;
  procedure WriteData;
  procedure ReadData;
public
  constructor Create(Name, DictFile: String);
  destructor Destroy; override;
  function AddClass(ClassName: String): TClass;
  procedure ProcessObjectRef;
  procedure Open;
  procedure Close;
  function ClassByName(ClassName: String) : TClass;
  function GetNewOID: TOID;
  property Name: String read FName;
  property Classes[Index: Integer]: TClass read GetClass;
  property ClassesCount: Integer read GetClassesCount;
  property ObjectClasses[Index: Integer]: TObjectRefType read GetObjectClasses;
  property ObjectClassesCount: Integer read GetObjectClassesCount;
end;

TClass = class
protected
  FDictOwner: TDatabase;
  FClassOwner: TClass; //Ponteiro da instancia através da qual uma outra instancia foi
                       //criada se FIsInstance=False é a própria classe SELF.
  FName: String;
  FAttributeList,
  FMethodList,
  FKeyList: TList;
  FBuffer: Pointer;
  FIsPersistent,
  FIsInstance: Boolean;
  FOID: TOID;
  FSuperClass: TClass;
  FInstancePos: LongInt; //Indica a posição onde a chave foi encontrada através da function
                         //BTBuscaComPilha
  FBinaryFile: TClassBin;

  //BTree OID Inicio
  FBTOIDStack: TList;
  BTOIDRoot: TBNode;

  //BTree Chave Inicio
  FBTStack: TList;
  BTRoot: TBNode;
  procedure BTStackInsert(Node: TObject; Index: Integer);
  procedure BTStackRemove(var Node: TObject; var Index: Integer);
  function BTBuscaComPilha(ChaveProc: Pointer): Boolean;
  procedure FreeBTree;
  //BTree Chave Fim

  //BTree OID Inicio
  procedure BTOIDStackInsert(Node: TObject; Index: Integer);
  procedure BTOIDStackRemove(var Node: TObject; var Index: Integer);
  function BTOIDBuscaComPilha(OIDProc: Pointer): Boolean;
  procedure FreeBTreeOID;
  //BTree OID Fim

  procedure VerifyBinary;
  procedure VerifyObjectRef;
  procedure VerifyDeleteObjectRef;

  procedure AllocBuffer;
  procedure FreeBuffer;
  procedure SetAttributeAddress;
  function GetSizeOfBuffer: Integer;
```

```
  function GetSizeOfKey: Integer;

  function GetAttributes(Index: Integer): TAttribute;
  function GetAttributesCount: Integer;
  function GetMethods(Index: Integer): TMethod;
  function GetMethodsCount: Integer;
  function GetKeyAttributes(Index: Integer): TAttribute;
  function GetKeyAttributesCount: Integer;

  procedure AddDefaultMethods;

public
  constructor Create(Name: String); virtual;
  destructor Destroy; override;
  procedure AddSuperClass(PSuperClass : TClass);
  function AddAttribute(Name: String; AttributeType: TAttributeType;
                        Width, Prec: Byte): TAttribute;
  function AddMethod(Name: String; MethodType: TMethodType; Attribute: String) : TMethod;
  function AddObject(Name, Reference: String): TAttribute;
  procedure AddKeyAttribute(AttributeName: String);
  function AttributeByName(AttributeName: String): TAttribute;
  procedure ClearAttributeValues;
  function MethodByName(MethodName: String): TMethod;
  function GetKeyBuffer: Pointer;
  function CreateKeyCopy: TList;
  procedure DestroyKeyCopy(Attributes: TList);
  function CompareKey(Key: TList; Operator: TCompOperator): Boolean;
  procedure SetPersistent;
  property Name: String read FName;
  property Attributes[Index: Integer]: TAttribute read GetAttributes;
  property AttributesCount: Integer read GetAttributesCount;
  property Methods[Index: Integer]: TMethod read GetMethods;
  property MethodsCount: Integer read GetMethodsCount;
  property KeyAttributes[Index: Integer]: TAttribute read GetKeyAttributes;
  property KeyAttributesCount: Integer read GetKeyAttributesCount;
  property IsPersistent: Boolean read FIsPersistent;
  property OID: TOID read FOID;
  property SuperClass: TClass read FSuperClass;
  //BTree Chave Inicio
  procedure BTInsere(ChaveIns: Pointer; Posicao: Longint);
  procedure BTPercorre;
  procedure BTRemove(ChaveRem: Pointer);
  //BTree Chave Fim
  //BTree OID Inicio
  procedure BTOIDInsere(OIDIns: Pointer; Posicao: Longint);
  procedure BTOIDPercorre;
  procedure BTOIDRemove(OIDRem: Pointer);
  //BTree OID Fim
  function InstanceCreate(PGetOid: Boolean): TClass;
  procedure InstanceDestroy;
  procedure PersistInstance;
  procedure FindInstance(PKeyValues: TStringList); overload;
  procedure FindInstance(POID: TOID); overload;
  procedure First;
  procedure Last;
  procedure Next;
  procedure Prior;
  function EOF: Boolean;
  function BOF: Boolean;
  function InstanceCount: Integer;
end;

TAttribute = class
protected
  FOwnerClass: TClass;
  FBuffer: Pointer;
  FName: String;
  FType: TAttributeType;
  FWidth,
  FDec: Byte;
  FReference: String;
  FIsInherited,
  FIsKey: Boolean;
```

```
  procedure SetAsString(Value: String);
  function GetAsString: String;
  procedure SetAsInteger(Value: Integer);
  function GetAsInteger: Integer;
  procedure SetAsDate(Value: TDateTime);
  function GetAsDate: TDateTime;
  procedure SetDecimal(Value: Integer);
  function GetDecimal: Integer;
  procedure SetAsObject(Value: TOID);
  function GetAsObject: TOID;
public
  constructor Create(Name: String; AttributeType: TAttributeType; Width, Prec: Byte);
  constructor CreateFrom(Source: TAttribute);
  function Size: Integer;
  function StrSize: Integer;
  function StrValue: String;
  property Name: String read FName;
  property AttributeType: TAttributeType read FType;
  property Width: Byte read FWidth;
  property Dec: Byte read FDec;
  property IsKey: Boolean read FIsKey;
  property IsInherited: Boolean read FIsInherited;
  property ReferenceClass: String read FReference;

  property AsString: String read GetAsString write SetAsString;
  property AsInteger: Integer read GetAsInteger write SetAsInteger;
  property Decimal: Integer read GetDecimal write SetDecimal;
  property AsDate: TDateTime read GetAsDate write SetAsDate;
  property AsObject: TOID read GetAsObject write SetAsObject;
end;

TMethod = class
protected
  FOwnerClass: TClass;
  FName: String;
  FType: TMethodType;
  FAttributeRef: String;
  FIsInherited : Boolean;

public
  constructor Create(Name: String; MethodType: TMethodType; PAttributeRef : String);
  procedure Execute(var PValue: Pointer; PKeyValues: TStringList = nil);
  property Name: String read FName;
  property MethodType: TMethodType read FType;
  property AttributeRef: String read FAttributeRef;
  property IsInherited: Boolean read FIsInherited;
end;

// Classes de acesso a arquivo binário
TClassHeader = record
               InstSize,
               KeySize,
               InstCount,
               FirstInst,
               LastInst: Longint;
               end;

TFileBin = class(TFileStream)
protected
  FBuffer: Pointer;
  FFileName: String;
  FHeader: TClassHeader;
  FBOF, FEOF: Boolean;

  procedure ReadFileHeader;
  procedure WriteFileHeader;
public
  constructor Create(FileName: String; InstSize, KeySize: Integer); virtual;
  destructor Destroy; override;

  procedure InsertInstance(BufInstance, BufKey: Pointer; POID: TOID = 0); virtual; abstract;
  procedure DeleteInstance; virtual; abstract;
  procedure UpdateInstance(BufInstance, BufKey: Pointer); virtual; abstract;
  procedure First; virtual; abstract;
```

```
  procedure Last; virtual; abstract;
  procedure Next; virtual; abstract;
  procedure Prior; virtual; abstract;
  procedure ReadInstance(Pos: Integer; PosOID: Integer = 0); virtual; abstract;

  property FileName: String read FFileName;
  property InstSize: Longint read FHeader.InstSize write FHeader.InstSize;
  property KeySize: Longint read FHeader.KeySize write FHeader.KeySize;
  property InstanceCount: Longint read FHeader.InstCount;
  property Buffer: Pointer read FBuffer;
  property EOF: Boolean read FEOF;
  property BOF: Boolean read FBOF;
end;

TIndexBin = class(TFileBin)
protected
  FOwner: TClassBin;
  FTypeIndex: TTypeIndex;
  function GetTotalSize: Integer;
  function InstPosition: Pointer;
  function PriorInstance: Pointer;
  function NextInstance: Pointer;
public
  constructor Create(FileName: String; InstSize, KeySize: Integer); override;
  procedure InsertInstance(BufInstance, BufKey: Pointer; POID: TOID = 0); override;
  procedure DeleteInstance; override;
  //procedure UpdateInstance(BufInstance, BufKey: Pointer); override;
  procedure ReadInstance(Pos: Integer; PosOID: Integer = 0); override;
  procedure First; override;
  procedure Last; override;
  procedure Next; override;
  procedure Prior; override;
end;

TClassBin = class(TFileBin)
protected
  FOwner: TClass;
  FKeyIndex,
  FOIDIndex: TIndexBin;
  FDataBuffer,
  FOIDBuffer : Pointer;
  function GetTotalSize: Integer;
  procedure CreateBTree;
  procedure SetBuffersAddress;
public
  constructor Create(FileName: String; InstSize, KeySize: Integer); override;
  destructor Destroy; override;
  procedure InsertInstance(BufInstance, BufKey: Pointer; POID: TOID = 0); override;
  procedure DeleteInstance; override;
  procedure UpdateInstance(BufInstance, BufKey: Pointer); override;
  procedure ReadInstance(Pos: Integer; PosOID: Integer = 0); override;
  procedure First; override;
  procedure Last; override;
  procedure Next; override;
  procedure Prior; override;
end;
```

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN; Jeffrey D. **Compiladores, princípios, técnicas e ferramentas.** Rio de Janeiro : Livros Técnicos e Científicos, 1995.

BAEHR Jr., Ivo. **Protótipo de sistema para gerenciamento de ordens de serviço acessando um banco de dados orientado a objetos e um banco de dados relacional**. 1999. 21 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

DATE, C.J. **Introdução a sistemas de banco de dados**. Rio de Janeiro : Campus, 1991.

GREIN, Dirceu; TOMIFA, Kátia Francelino. **Sistemas de gerenciamento de banco de dados orientado a objetos**. Curitiba, 1998. Disponível em: <http://www.celepar.gov.br/batebyte/Internet-anteriores/1998/bb78/colunado.htm>. Acesso em: 24 fev. 2001.

KHOSHAFIAN, Setrag. **Banco de dados orientado a objeto**. Rio de Janeiro : Infobook, 1994.

KORTH, Henry F.; SILBERSCHATZ, Abraham. **Sistema de bancos de dados.** São Paulo : Makron Books, 1995.

MARTIN, James; ODELL, James J.. **Análise e projeto orientados a objeto.** São Paulo : Makron Books, 1995.

NASSU, Eugênio A.; SETZER, Valdemar W. **Bancos de dados orientados a objetos.** São Paulo : Edgard Blücher, 1999.

RAO, Bindu Rama. **Object-oriented databases**. New York : McGraw-Hill, 1994.

RUMBAUGH, James et al. **Modelagem e projeto baseados em objetos.** São Paulo : Campus, 1994.

SETZER, Valdemar W.; Melo, Inês S. Homem de. **A construção de um compilador.** Rio de Janeiro : Campus, 1985.

VILLAS, Marcos Vianna et al. **Estrutura de dados:** conceitos e técnicas de implementação. Rio de Janeiro : Campus, 1993.