UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL ATRAVÉS DO REPOSITÓRIO DA FERRAMENTA CASE SYSTEM ARCHITECT

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

DANILO KRAMEL

BLUMENAU, DEZEMBRO/2000

2000/2-16

PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL ATRAVÉS DO REPOSITÓRIO DA FERRAMENTA CASE SYSTEM ARCHITECT

DANILO KRAMEL

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. Maurício Capobianco Lopes

AGRADECIMENTOS

Gostaria de aproveitar a oportunidade para agradecer aos meus pais Daniel Kramel e Lourdes Marta Kramel pelo incentivo para estudar, pelo carinho e dedicação.

Ao professor Marcel Hugo, que me orientou neste trabalho com paciência e sabedoria. Aos professores Everaldo Artur Grahl e José Roque Voltolini da Silva pela contribuição que deram para a realização deste trabalho.

Também gostaria de agradecer ao Ivo Baehr Junior, conhecedor da tecnologia de banco de dados Cachê e Ivete Militzer, gerente de negócios da Choose Technologias - empresa responsável pela ferramenta *System Architect*, que mesmo não nos conhecendo pessoalmente não mediram esforços quando necessitei de ajuda referente a parte que lhes cabia.

E também gostaria de agradecer a minha namorada, meus irmãos, meus amigos e todas as pessoas que de uma forma direta ou indireta contribuíram para a realização deste trabalho.

A todos muito obrigado.

SUMÁRIO

RESUMO

Este trabalho consiste na pesquisa sobre a área de banco de dados orientado a objetos e o auxílio de ferramenta CASE para a modelagem de sistemas. Obteve-se como resultado a implementação de um protótipo para a geração de código *Class Definition Language* (CDL) para banco de dados Caché a partir do repositório da ferramenta CASE *System Architect* (SA).

ABSTRACT

This work consists of the research on the object oriented database and the tool aid CASE for the modeling systems. The result, is the implementation of a prototype for the code generation to Class Definition Language (CDL) for database Caché starting from the repository of the tool CASE System Architect (SA).

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

BDOO - Banco de dados Orientado a Objetos

CAD - *Computer Aided Design*

CAM - *Computer Aided Manufacture*

CASE - *Computer Aided Software Engineering*

CDL - *Class Definition Language*

DBA - *DataBase Administrator*

DBF - *Dbase Format*

SA - *System Architect*

SGBD - Sistemas de Gerenciamento de Banco de Dados

SQL - *Structured Query Language*

UML - *Unified Modeling Language*

1 INTRODUÇÃO

1.1 ORIGEM

Apesar da tecnologia da orientação a objetos ter surgido nos anos 70, somente nos últimos anos presencia-se o renascimento da mesma em todos os segmentos da computação, onde vem tornando-se uma disciplina de desenvolvimento de software viável e utilizada na maioria dos projetos de larga escala ([KHO1994]).

As técnicas orientadas a objetos mudam a visão que os analistas de sistema de informação têm do mundo ([MAR1995]). A tecnologia de objetos apresenta componentes chaves que fundamentam a mudança de enfoque no processo de modelagem e desenvolvimento de aplicações, trazendo benefícios intrínsecos à filosofia. Contudo, essa mudança surgiu devido à necessidade da indústria de software possuir tecnologia mais avançada para o desenvolvimento de sistemas mais complexos com maior facilidade ([FUR1998]).

Paralelo ao surgimento dos sistemas orientados a objetos, um fator igualmente importante tem sido a evolução dos sistemas de gerenciamento de bancos de dados orientado a objetos que são baseados na tecnologia de programação orientada a objetos ([KHO1994] [KOR1991]). O banco de dados orientado a objetos segundo [MAR1995], "... é um banco de dados inteligente. Ele suporta o paradigma orientado a objeto, armazenando dados e métodos em vez de apenas dados. Ele é projetado para ser fisicamente eficiente para armazenar objetos complexos e permite o acesso aos dados por meio dos métodos armazenados. "

Perante esse renascimento da orientação a objeto surgiu também a utilização de ferramentas CASE (*Computer Aided Software Engineering*) orientadas a objetos. Para [MAR1995], as técnicas de orientação a objetos e a tecnologia CASE convivem naturalmente e formam um poderoso mecanismo para desenvolvimento de software.

Contudo, as indústrias de software não iriam mudar de paradigma se a orientação a objetos não trouxesse benefícios para as mesmas. Para [WIN1993], os benefícios primários referem-se a habilidade de administrar a complexidade dos sistemas e o aumento de produtividade no processo de desenvolvimento.

Segundo [MAR1995], o mundo orientado a objetos é mais disciplinado do que as técnicas estruturadas convencionais. Ele leva a um mundo de classes reusáveis no qual grande parte do processo de construção de software será a montagem de classes existentes bem comprovadas, onde os benefícios da orientação a objetos vinculados com ferramentas CASE orientadas a objetos baseados em repositórios, podem ser: reusabilidade, estabilidade, confiabilidade, integridade e manutenção mais fácil.

Desta forma, os bancos de dados orientados a objetos surgiram com o objetivo de unir os conceitos e benefícios das tecnologias de programação orientada a objetos e banco de dados, proporcionando aos desenvolvedores um mecanismo mais eficiente para desenvolver sistemas mais complexos.

Entre os vários banco de dados orientados a objetos pode-se destacar o Caché, produto da InterSystems Corporation. Esse possui, como alternativa de manipulação de suas classes, através de arquivos texto no formato ASCII, o *Class Definition Language* (CDL). Os arquivos CDL são muito úteis quando deseja-se exportar ou importar classes e definições entre um sistema Caché e outro. Quando os arquivos CDL são importados, eles são compilados pelos programas do API *ClassDictionary*, construindo assim, toda a estrutura de definição das classes ([BAE1999]).

1.2 OBJETIVOS

O objetivo principal deste trabalho consiste em desenvolver um protótipo de software de geração de código CDL (*Class Definition Language*) para banco de dados Caché a partir do repositório da ferramenta CASE *System Architect*.

Os objetivos secundários desse trabalho são:

a) pesquisar sobre a área de banco de dados orientado a objetos (banco de dados Caché);
b) aprofundar os conhecimentos em CASE (*System Architect*).

1.3 ORGANIZAÇÃO DO TEXTO

No capítulo 1 tem-se a introdução do trabalho, juntamente com o seu objetivo.

No capítulo 2 é mostrado para a história e evolução dos bancos de dados e também os conceitos que os banco de dados orientados a objetos.

No capítulo 3 é mostrado os conceitos e finalidades da utilização de ferramentas CASE.

No capítulo 4 a ferramenta CASE *System Architect* é apresentada, incluindo sua características principais e como ela administra as informações de seus modelos, na forma de repositório.

No capítulo 5 é apresentado o banco de dados Caché, incluindo suas características, estrutura e a *Class Description Language* (CDL) que é utilizada para a geração de código.

No capítulo 6 é apresentando as fases que um compilador possui e a semelhança com o protótipo.

No capítulo 7 é apresentado o protótipo de software de geração de código *Class Definition Language* (CDL) para banco de dados Caché a partir do repositório da ferramenta CASE *System Architect* (SA) juntamente com sua especificação, passos para a sua execução e estudo de caso.

No capítulo 8 é apresentada a conclusão, assim como sugestões para a continuação deste trabalho.

2 BANCO DE DADOS ORIENTADO A OBJETOS

2.1 CONCEITUAÇÃO DE BANCO DE DADOS

A área de tecnologia de informação, desde os primórdios, esteve baseada no armazenamento e disponibilidade de informações. No início, as informações estavam contidas em arquivos seqüências e indexados. O desenvolvimento era dirigido para setores específicos da empresa, possuindo dados e arquivos utilizados somente por aquele setor afetando gradativamente os outros setores ([LEI1980]).

Dessa forma, os desenvolvedores tinham problemas/dificuldades no controle de redundância de dados, garantia de integridade dos dados, dificuldades na manutenção e na criação de novas aplicações, pois as linguagens utilizadas não incorporavam todas as funcionalidades desejadas, sendo necessário desenvolvê-las. Com o tempo, verificou-se que essas operações eram comuns em muitos sistemas e que havia a necessidade de evolução das ferramentas para o desenvolvimento de sistemas ([HEU2000] [LEI1980]).

Durante a década de 70, surgiu uma nova técnica para solucionar os problemas referentes ao armazenamento de informações, conhecida como banco de dados ([URR1999]). Para [RAM1997], um banco de dados é uma coleção de dados, que descreve as atividade de relacionamento entre uma ou mais entidades.

Segundo [DAT1991], um sistema de banco de dados é responsável por armazenar qualquer informação considerada como importante ao indivíduo ou à organização servida pelo sistema e torná-la disponível quando solicitada.

Dessa forma, um banco de dados possui uma arquitetura geral, que encontra-se dividido em três níveis:

a) nível interno/físico: é o nível mais próximo ao armazenamento físico, especificando os detalhes de organização e estrutura do arquivo;
b) nível externo/lógico: é o mais próximo ao usuário, é nele que se descreve a estrutura das entidades e com elas se relacionam;
c) nível conceitual: é o responsável pelo intercâmbio entre o nível interno e externo.

Além dos níveis de arquitetura que o banco de dados possui, existem vários modelos. Para [HEU2000], um modelo de banco de dados diz respeito à descrição formal de sua estrutura.

Dentre os modelos de dados existentes, quatro destacaram-se, são eles:

a) Relacional;
b) Hierárquico;
c) Rede;
d) Orientado a objeto.

Maiores informações sobre os modelos de banco de dados, podem ser encontradas em: [BAE1999], [DAT1991] e [KHO1994].

Um ponto importante a ser considerado sobre banco de dados, diz respeito ao sistema de gerenciamento de banco de dados (SGBD), responsável por administrar as informações no banco de dados. Segundo [HEU2000], um SGBD é um software que incorpora as funções de definição, recuperação e alteração de dados em um banco de dados.

Já para [KHO1994], um SGBD permite a um banco de dados persistente ser concorrentemente partilhado por muitos usuários e aplicativos. E para alcançar esse objetivo com eficiência, os SGBD usam controle de concorrência subjacente, manuseio de armazenamento e estratégias de otimização.

A utilização de banco de dados tem várias vantagens, como ([RAM1997] [LEI1980] [HEU2000]):

a) controle de redundância de dados: um SGBD elimina o problema de inconsistência dos dados, pois possui um controle centralizado das diversas aplicações que utilizam o banco de dados;
b) garantia de integridade dos dados: quando dois ou mais usuários utilizam o mesmo dado ao mesmo tempo, o SGBD garante que os dados não serão perdidos;
c) privacidade dos dados: o SGBD permite que sejam estabelecidas chaves de segurança contra acessos não autorizados;

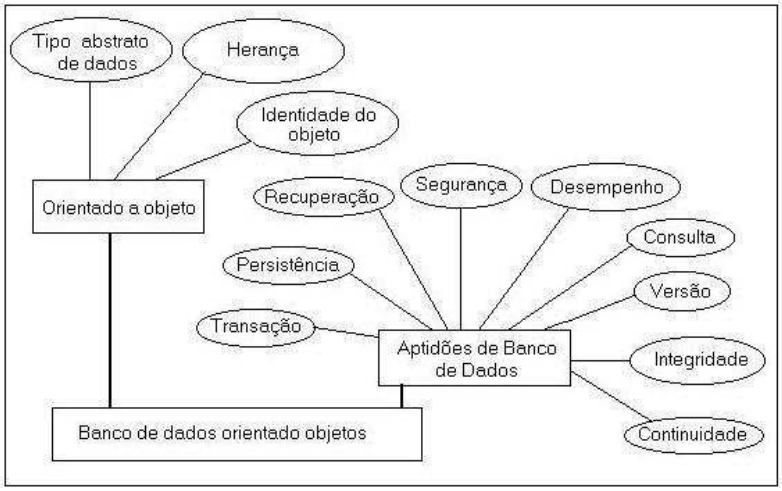
d) facilidade de criação de novas aplicações: um sistema de banco de dados garante uma independência dos controles de armazenamento em relação ao desenvolvimento de novas aplicações;
e) eficiência no acesso aos dados: o sistema de banco de dados utiliza uma variedade de tecnologias para armazenar e recuperar informações eficientemente.

2.2 SURGIMENTO DO BANCO DE DADOS ORIENTADO A OBJETOS (BDOO)

Os primeiros bancos de dados orientados a objetos surgiram em universidades e centros de pesquisas, em meados dos anos 80, com o objetivo de sustentar a programação orientada a objeto, pois os programadores de linguagens orientadas a objetos precisavam de um depósito para os objetos permanentes (objetos que permanecem depois que o processo é encerrado) ([MAR1995], [TAU1997]).

Para [KOR1994], banco de dados orientado a objetos é a união da tecnologia de banco de dados com a da orientação a objetos, conforme figura 01.

FIGURA 1 - BANCO DE DADOS ORIENTADO A OBJETOS



Fonte : [KOR1994]

Comercialmente, eles surgiram como gerenciadores de objetos embutidos em aplicações de engenharia de projeto auxiliado por computador (CAD – *Computer-Aided Design),* manufatura auxiliada por computador (CAM – *Computer-Aided Manufacture*) e engenharia de software auxiliada por computador (CASE). Maiores informações de como os

BDOO contribuíram para as aplicações de engenharia CAD, CAM e CASE podem ser adquiridas em: [BAE1999], [KOR1994] e [WIN1993].

A necessidade comum dessas aplicações em utilizar as funcionalidades de banco de dados orientado a objetos, diz respeito à manipulação de tipos de dados complexos e captura de estruturas dos dados. Se a informação que está sendo buscada no banco pode ser entendida isoladamente, o banco de dados relacional é a melhor escolha, mas se a informação só faz sentido no contexto de outros registros, o banco de dados orientado a objetos é mais apropriado ([WIN1993]).

Muitas outras aplicações partilham da necessidade de um banco de dados mais poderoso, no qual os dados sejam processados juntamente com informações gráficas, sons e textos. Dentre essas, pode-se destacar aplicações para WEB, multimídia e escritórios inteligentes ([KOR1994] [TAU1997] [WIN1993]).

Para [BAE1999], os banco de dados tradicionais esbarram em uma série de problemas quando existe a necessidade de gerenciar aplicações mais complexas:

a) falta de um tecnologia mais avançada para a modelagem de dados: as aplicações tradicionais possuem estruturas limitadas se analisadas com aplicações mais complexas. Dessa forma, os benefícios que a tecnologia de orientação a objetos oferece não são suportados pelos bancos de dados tradicionais;
b) falta de mecanismo para suportar transações longas: em bancos de dados tradicionais, os dados permanecem bloqueados durante o tempo de manutenção. Isso ocasiona um problema para aplicações de próxima geração, pois elas podem passar dias efetuando uma alteração, sendo inadmissível que os dados permaneçam bloqueados todo esse tempo;
c) desigualdade de impedância: diz respeito a diferença com que o banco de dados e linguagens de programação podem suportar modelos de dados e paradigmas de objetos;
d) problemas de desempenho: devido à complexidade dos modelos de objetos, as transações que os envolvem diferem das atuais suportadas pelos bancos de dados tradicionais. Em bancos de dados tradicionais as buscas e atualizações em modelos

de entidades e relacionamento podem ser consideradas simples se comparadas com as pesquisas em modelos de objetos utilizando banco de dados relacionais;

e) dados comportamentais: facilidade/benefício que a tecnologia de orientação a objetos disponibiliza para que objetos diferentes possam responder de formas diferentes ao mesmo comando;

f) falta de suporte à evolução de esquemas e versões: a maioria dos bancos de dados permitem que somente uma representação de uma entidade exista, e qualquer evolução ou alteração no esquema é uma tarefa árdua a ser desenvolvida pelo *DataBase Administrador* (DBA). Em aplicações de próxima geração essas atividades podem tornar-se corriqueiras e as operações não podem ser complexas.

2.3 CONCEITOS DE ORIENTAÇÃO A OBJETOS APLICADOS EM BANCO DE DADOS

Para [WIN1993], os BDOO oferecem algumas das mesmas funcionalidades das linguagens orientadas a objetos. Eles permitem encapsular dentro dos objetos, os dados e métodos que atuam sobre os objetos. Ativam métodos por mensagens aos objetos e permitem declarações de relacionamentos hierárquicos entre objetos por meio do uso de hereditariedade.

Mas em BDOO a noção de objeto é usada somente no nível lógico e possui características não encontradas nas linguagens de programação tradicionais de orientação a objetos. Estas características dizem respeito a operadores de manipulação de estruturas, gerenciamento de armazenamento, tratamento de integridade e persistência dos dados ([SAL1992]).

Segundo [SAL1992], os principais conceitos da orientação a objetos aplicados aos BDOO são: objetos complexos, identidade de objeto, encapsulamento, classes, herança e extensibilidade.

2.3.1 OBJETOS DE BANCO DE DADOS

Segundo [MAR1995], “um objeto é qualquer coisa, real ou abstrata, a respeito da qual armazenamos dados e os métodos que os manipulam.”

Para [WIN1993], objetos são módulos que contém dados e instruções para operar sobre estes dados, possuindo características das linguagens tradicionais, como números, matrizes, strings e registros, bem como funções, instruções ou sub-rotinas.

Já para [SAL1992], os objetos são abstrações de dados do mundo real, com uma interface de nomes de operações e um estado local que permanece oculto. Possuindo um estado interno descrito por atributos que podem ser acessados ou modificados apenas pelas operações definidas pelo criador do objeto.

O termo objeto aplicado a banco de dados é similar à noção de entidade dos bancos de dados relacionais, pois ambos preocupam-se em armazenar os dados. Mas nos objetos, são acrescentados os métodos que são responsáveis por manipular os dados/atributos dos objetos.

2.3.1.1 IDENTIDADE DO OBJETO

Segundo [BAE1999], a identidade do objeto corresponde, em geral, ao endereço físico de sua instância. Esse endereço em linguagens orientadas a objetos é criado em tempo de execução e destruído no final da execução.

Para [KOR1994], a identidade do objeto é uma propriedade de um objeto que distingue cada objeto dos outros, permitindo assim, que o objeto possa ser referenciado por sua identidade.

Em BDOO o conceito de identidade do objeto deve ser mais forte do que o aplicado nas linguagens orientadas a objetos. Essa necessidade acontece porque os objetos em BDOO devem continuar existindo mesmo após a execução do programa ([NAS1999]).

A identidade do objeto deve ser única dentro do banco de dados e não pode ser comparada com o conceito de chave primária dos bancos de dados relacionais. A chave primária em banco de dados relacionais é mutável e aplicada somente na entidade, ao contrário dos BDOO que a identidade do objeto é aplicada em todo o banco de dados e única desde a criação do objeto até a sua destruição ([SAL1992]).

O uso de identificadores únicos de objetos elimina anomalias de atualização e de integridade referencial, além de tornar mais fácil a referência e as associações de objetos.

Qualquer atualização nos valores dos atributos que um objeto vier a sofrer, não causará impacto nos objetos que os referenciam ([BAE1999] e [SAL1992]).

2.3.1.2 OBJETO COMPLEXO

Segundo [MAR1995], objetos complexos são objetos que são compostos de outros objetos ou fazem referencia a objetos.

Para [SAL1992], objetos complexos são formados por construtores (conjuntos, listas, registros, *arrays*) aplicados a objetos simples (inteiros, booleanos, *strings*). Onde a manutenção desses objetos complexos, independentemente de sua composição, requer a definição de operadores apropriados para sua manipulação.

Segundo [BAE1999], existe basicamente dois tipos de objetos complexos em BDOO:

a) objetos embutidos: são objetos que estão embutidos em forma de atributos em um objeto maior, conhecido como agregação ou todo-parte. Os objetos embutidos não possuem identificadores e são armazenados na mesma estrutura física do objeto superior;
b) objetos referenciados: ocorre quando existe relacionamento entre os objetos. Neste caso, todos os objetos possuem um identificador próprio e podem ser acessados diretamente ou através dos objetos relacionados.

2.3.2 ENCAPSULAMENTO

Segundo [MAR1995], encapsulamento é a possibilidade do objeto permitir esconder seus dados de outros objetos, permitindo que somente sejam acessados por intermédio de seus próprios métodos.

Já para [SAL1992], o encapsulamento possibilita a distinção entre a interface do objeto visível ao aplicativo e a implementação das operações. Desta forma, provê uma modularidade que permite uma melhor estruturação das aplicações ditas complexas, bem como a segurança dentro do sistema.

O encapsulamento é importante porque provê uma forma de independência lógica dos dados, separando a maneira como um objeto se comporta da maneira como ele é

implementado. Dessa forma, a implementação do objeto pode ser alterada sem exigir que os aplicativos que a utilizam sejam modificados, além de prover vantagens escondendo a complexidade do código e evitar que os dados sejam corrompidos por aplicações externas([MAR1995], [ROC1996] e [WIN1993]).

Em banco de dados diz-se que um objeto está encapsulado quando os dados são ocultos aos usuários e o objeto pode ser consultado e modificado exclusivamente por meio das operações a ele associado. O código destas operações permanece oculto ao usuário, que é acessado apenas através da interface ([SAL1992]).

2.3.3 CLASSES

[MAR1995] especifica o termo classe como sendo a implementação de um tipo de objeto, determinando a estrutura de dados e os métodos operacionais que permitem o acesso e modificação dos dados do objeto

Um conjunto de objetos que possui os mesmos atributos, relacionamentos e operações, podem ser agrupados para formar uma classe, visto que as classes não possuem um estado e nem um comportamento específico, mas uma forma que permite criar novos objetos ([ROC1966], [SAL1992]).

Segundo [SAL1992] as classes possuem característica de “fábrica” ou de “depósito” de objetos. A fábrica é utilizada para criar novos objetos associados à classe. Já o depósito mostra o conjunto de objetos que estão associados à classe.

As classes pré-definidas dos BDOO diferem das linguagens orientadas a objetos, principalmente pelos tipos de classes. Ambas, tipicamente incluem classes para tipos básicos de dados, como inteiros, reais, *strings*, conjuntos e matrizes. Mas normalmente em BDOO, são incluídas classes para objetos do tipo persistentes, exceções, diretórios, bloqueios, resumos e outros tipos de banco de dados ([WIN1993]).

2.3.4 HERANÇA

Segundo [SAL1992], a herança é um mecanismo que permite definir tipos de forma incremental, por refinamento de outros já existentes, permitindo a construção de tipos em que

as propriedades (atributos, métodos), de um ou mais tipos são reutilizadas na definição de um novo tipo.

[GON2000] determina herança como um espécie de relacionamento entre classes, onde um classe herda atributos e métodos de outras classes. As vantagens são prover uma maior expressividade na modelagem dos dados, facilitar a reusabilidade de código e definir classes por refinamento.

Existem dois mecanismos de herança, conhecidos como simples e múltipla. Com a herança simples, uma subclasse pode herdar dados e métodos de uma única classe, além de acrescentar ou subtrair comportamento para si própria. A herança múltipla refere-se à habilidade de uma subclasse adquirir dados e métodos de várias classes ([WIN1993])

Em banco de dados o conceito de herança é aplicado para os atributos, relacionamentos e operações ([SAL1992]).

2.3.5 EXTENSIBILIDADE

Todos os bancos de dados possuem tipos pré-definidos, como inteiros, reais e *strings*. Nos BDOO, as novas classes criadas pelo usuário não podem ter diferença de tratamento em relação aos tipos pré-definidos. Isso é necessário, pois permite que as novas classes possam ser utilizadas semelhante aos tipos pré-definidos ([NAS1999]).

2.4 CONCEITOS DE BANCO DE DADOS APLICADOS A BANCO DE DADOS ORIENTADOS A OBJETOS

A finalidade de um banco de dados independente de ele ser orientado a objetos ou não, é a de armazenar dados relevantes para a entidade de forma que possam ser recuperados quando solicitados. Desta forma, existem conceitos que são aplicados em ambas as tecnologias (banco de dados tradicionais e banco de dados orientado a objetos), mas adaptados às necessidades e realidades individuais.

2.4.1 PERSISTÊNCIA DE OBJETOS

No contexto de banco de dados o termo “persistência” raramente é utilizado, preferencialmente o termo usado é “banco de dados”, que conota o espaço de dados maleáveis e concorrentemente compartilhados, sendo uma das funções do SGBD, permitir o acesso e a atualização simultânea dos dados persistentes ([KHO1994]).

Para [WIN1993], no contexto de linguagens orientadas a objetos, persistência refere-se à capacidade do objeto permanecer acessível na memória enquanto o programa é executado.

Segundo [KHO1994], existem vários níveis de persistência, determinadas de acordo com a necessidade do sistema, podendo ser classificada de uma forma geral, como sendo local, de sessão ou único. Nos dois primeiros níveis (local, sessão), a diferença consiste no momento em que os objetos estão disponíveis, podendo ser em uma rotina específica ou em módulo, mas ambos são perdidos quando a aplicação é encerrada.

No terceiro nível (único), refere-se à possibilidade do objeto continuar existindo depois que a aplicação foi encerrada. Essa capacidade de persistência está amarrada ao conceito de identidade do objeto, a qual exige que o objeto tenha um identificador que seja único e invariável ([WIN1993]).

No BDOO o objeto pode continuar existindo mesmo após o encerramento do programa, tendo seu estado armazenado em um meio físico persistente ([NAS1999]).

2.4.2 CONTROLE DE TRANSAÇÃO

Segundo [HAC1993], uma transação é uma ou mais ações de banco de dados que são tratadas como uma unidade simples de trabalho.

Para melhor entender o que é uma transação, existe um conjunto de quatro propriedades freqüentemente chamadas de ACID: Atomicidade, Consistência, Isolamento e Durabilidade ([HAC1993], [KHO1994]):

a) atomicidade: é uma transação que deve ser executada inteiramente ou não executada. O sistema de banco de dados deve garantir que todas as operações realizadas por uma transação concluída com sucesso sejam refletidas no banco de

dados e que os efeitos de uma transação malsucedida sejam completamente desfeitas;

b) consistência: está relacionada a toda preservação das restrições semânticas do banco de dados, sendo considerado coerente se todas as restrições forem satisfeitas. Em outras palavras, quando a transação é comprometida a aplicação deve garantir que todos os dados afetados estão num estado consistente, evitando obviamente o estado inconsistente;
c) isolamento: trata da segurança fornecida pelo sistema de banco de dados em relação aos conflitos entre transações concorrentes. A transação deve ser executada isoladamente de outras transações, e uma vez que uma transação é iniciada, seu efeito deve ser o mesmo, desconsiderando quaisquer outras ações no banco de dados;
d) durabilidade: significa que as atualizações de transação efetivadas com sucesso nunca se percam, permitindo ser recuperadas no caso do sistema ou o meio apresentar falhas.

As transações em banco de dados convencionais são curtas e atualizam e referenciam somente alguns registros. Já em BDOO as transações possuem um papel mais complexo, podem ser mais demoradas ou necessitarem de trabalho em grupo para serem executadas ([BAE1999]).

Dessa forma, o modelo de transações aninhadas introduzido por Moss (1981), tem como objetivo resolver o problema das transações demoradas. Esse modelo é constituído de um estrutura de árvore, decompondo a transação principal em uma série de subtransações que devem ser executadas com sucesso para obter-se o resultado da transação principal ([BAE1999], [KHO1994]).

Já o modelo de transação em cooperação tem como objetivo resolver o problema de transações que necessitam trabalhar em conjunto para serem concluídas. O contraste entre as transações tradicionais e as transações em cooperação está na possibilidade de ver os resultados imediatos umas das outras ([BAE1999]).

2.4.3 CONTROLE DE CONCORRÊNCIA

Segundo [HAC1993], a concorrência significa que várias atividades estão acontecendo na mesma unidade de tempo, mas não necessariamente no mesmo momento. Desta forma, vários problemas podem ser verificados no processo de transações concorrentes, como leitura suja, leitura não repetível, linhas fantasmas e perda de atualização.

Nas implementações tradicionais, a garantia de integridade do banco de dados durante operações concorrentes, é geralmente baseada nas regras impostas a cada item de dados a ser lido ou gravado, tornando as operações possíveis ou não para uma ou outra transação ([WIN1993]).

Segundo [KHO1994], existe tradicionalmente três estratégias principais de controle de concorrência utilizadas:

a) algoritmo pessimista: pressupõem que transações concorrentes provavelmente conflitarão e portanto adquirem bloqueios antes de acessar e atualizar as operações;
b) algoritmo otimista: que as transações provavelmente operarão sem conflito e só na hora da efetivação são feitas as conferências para garantir o isolamento das transações;
c) versionamento: cria uma nova versão de um objeto para cada atualização.

Segundo [BAE1999], existem basicamente dois tipos de bloqueio. O primeiro, de leitura/compartilhamento, permite a várias transações realizarem leituras concorrentes no mesmo objeto. O segundo, de gravação/exclusividade, reserva o acesso das operações de leitura e gravação de um objeto à transação que solicitou o bloqueio. Nesse tipo de bloqueio nenhum outro bloqueio pode ser disponibilizado.

Maiores informações sobre os diversos tipos de bloqueios podem ser adquiridas em [KHO1994] e [HAC1993].

2.4.4 CONSULTAS

Os conceitos da orientação a objetos requerem que toda a interação com o objeto seja feita por meio de envio de mensagens. Em um BDOO, uma única mensagem origina muitas

consultas relacionadas, que podem solicitar cálculos e provocar mensagens a serem enviadas a outros objetos antes de retornar um resultado, aliviando o problema da necessidade de consultas seqüenciais ([KOR1995], [WIN1993]).

Dessa forma, em linguagens orientadas a objetos, deve-se enviar uma mensagem à cada instância da classe, para se obter a ocorrência de um determinado atributo. Já em BDOO isso pode ser resolvido através de um modelo de mensagem de objeto para conjuntos, onde é possível efetuar consultas que envolvem junções de conjuntos de objetos ([BAE1999]).

Assim, uma linguagem de consulta de BDOO precisa incluir tanto o modelo de passagem de mensagens de um objeto, como o modelo de passagem de mensagens de um conjunto de objetos ([KOR1995]).

2.4.5 VERSIONAMENTO

Segundo [KHO1994], o versionamento em um banco de dados orientado a objetos consiste em ferramentas e construções que automatizam ou simplificam a construção e a organização de versões.

Uma característica presente em alguns sistemas é o controle de versões de objetos. Em aplicações de engenharia como CAM ou CASE, pode ser necessário o armazenamento de versões anteriores dos projetos, tendo a possibilidade de retornar ao estado anterior caso ocorra algum problema/erro. Uma outra situação pode ocorrer quando houver modificação nos métodos ou atributos do objeto, sendo desejável manter uma versão estável e testada do objeto enquanto a nova versão não for aprovada ([NAS1999]).

Depois que um objeto é versionado, é criada uma estrutura que contém o conjunto das várias versões do objeto. Essa estrutura permite criar novas versões, contendo todas as características da versão de um objeto preexistente, tendo como propriedade comum a todas versões a identidade do objeto.

Segundo [BAE1999], o esquema de versionamento mais comum é a sucessão linear de gerações, onde os objetos são criados seqüencialmente, formando um conjunto de versões

lineares. Porém, existe casos em que são criadas versões em paralelo de um mesmo objeto, denominadas de sucessão alternativa de gerações.

3 FERRAMENTAS CASE

Perante o renascimento da orientação a objetos surgiu também a utilização de ferramentas CASE (*Computer Aided Software Engineering*) orientadas a objetos. Para [MAR1995], as técnicas de orientação a objetos e a tecnologia CASE baseada em repositório convivem naturalmente, formando um poderoso mecanismo para desenvolvimento de software.

3.1 HISTÓRIA

No decorrer dos últimos 20 anos, muitos engenheiros de software eram menosprezados dos recursos que a computação poderia oferecer a seu trabalho. Construído sistemas complexos que automatizava o trabalho para outros, mas muito pouco para si mesmo.

Até recentemente, a engenharia de software era fundamentalmente uma atividade manual em que as únicas ferramentas disponíveis aos engenheiros eram os compiladores e os editores de texto, representando não mais de 20% do processo de engenharia de software global ([PRE1995]).

Os engenheiros de software de todas as plataformas de hardware, desde os computadores pessoais até os de grande porte, esforçavam-se para satisfazer seus usuários com produtos cada vez mais inteligentes, robustos e potentes. Contudo, o crescimento e a complexidade cada vez maior, tornava-se propenso a um comportamento imprevisível e, como conseqüência, o fracasso ([FIS1990]).

Desta forma, as ferramentas CASE surgiram para reduzir substancialmente, ou eliminar, inúmeros problemas de desenvolvimento de projeto e sistemas, através das especificações dos analistas, possibilitando a geração automática de grande parte do sistema ([FIS1990]).

3.2 FINALIDADE

Segundo [PRE1995], a engenharia de software compreende um conjunto de três etapas (métodos, ferramentas e procedimentos) fundamentais, que possibilita ao analista o controle

do processo de desenvolvimento de um sistema, oferecendo ao profissional uma base para a construção de software de alta qualidade e produtividade.

Os métodos de engenharia de software proporcionam os detalhes de como fazer para construir o software, envolvendo um conjunto de tarefas que incluem: planejamento e estimativas de projeto, análise de requisitos, projeto de estrutura de dados, arquitetura de programas e algoritmos de processamento, codificação, teste e manutenção ([PRE1995]).

As ferramentas de engenharia de software proporcionam apoio automatizado ou semi - automatizado aos métodos. Dessa forma, quando as ferramentas são integradas de forma que a informação criada por ela possa ser usada por outra, estabelece-se um sistema de suporte ao desenvolvimento de software chamado “engenharia de software auxiliado por computador” (CASE - *Computer Aided Software Engineering*) ([PRE1995]) .

Os procedimentos da engenharia de software constituem o elo de ligação entre os métodos e as ferramentas, possibilitando o desenvolvimento racional e oportuno do software e definindo a seqüência em que os métodos serão aplicados ([PRE1995]).

Segundo [RAF2000], um produto é classificado como um ferramenta CASE quando este oferece documentação, automação, racionalização do projeto e implementação. Dentre essas classes de ferramentas, pode-se destacar os geradores de aplicações, as ferramentas de modelagem e as implementações de bancos de dados

[FIS1990] define CASE como sendo uma “ferramenta que oferece impulso em qualquer ponto do ciclo de desenvolvimento do software”. Incluindo assim a maioria das ferramentas conhecidas pelos engenheiros de softwares, inclusive os compiladores, depuradores, geradores de perfis de desempenho e sistemas de controle de código.

A utilização de ferramentas CASE na engenharia de software pode trazer várias vantagens como ([FIS1990]):

a) especificações completas dos requisitos: possibilita ao projetista de software especificar todos os requisitos do sistema, para evitar que o aplicativo final seja diferente daquilo que o usuário pretendia.

b) especificações minuciosas do projeto: permite a especificação de documentos que facilitam o entendimento do sistema.
c) especificações atuais do projeto: mantém uma sincronia com a especificação e o código, de modo que se ocorrer alguma modificação na especificação o código subordinado também se modifica.
d) redução do tempo de desenvolvimento: a especificação completa da arquitetura do software reduz substancialmente, quando não elimina, a perda de tempo com códigos desnecessários ou desperdiçados.
e) código altamente flexível e de fácil manutenção: permite uma maior facilidade no aperfeiçoamento ou correção que o sistema possa ter.

3.3 CATEGORIA

Segundo [MAR1995], as ferramentas CASE podem ser categorizadas como aquelas desenvolvidas para o mundo orientado a objeto, e aquelas desenvolvidas para o mundo da computação convencional.

Uma outra forma de categorizar as ferramentas, está relacionada a amplitude que ela oferece ao ciclo de vida do software, podendo ser categorizadas como ([MAR1995]):

a) IE-CASE: suporta toda a engenharia da informação;
b) I-CASE: refere-se a CASE integrado, mas não dando suporte a engenharia da informação;
c) *Fragment* CASE: são ferramentas fragmentadas, freqüentemente conhecidas como *front-end* para a análise e *back-end* para a geração de código.

Para [PRE1995], as ferramentas CASE podem ser classificadas por função, por utilidade aos gerentes e pessoas técnicas, por aplicabilidade nas etapas de processo de engenharia de software, pela arquitetura de ambiente (hardware e software), pela origem e custo. Abaixo será listada uma classificação de ferramentas por função:

a) ferramentas de planejamento de sistemas comerciais: o objetivo primordial dessa ferramenta é ajudar a melhorar a compreensão de como a informação flui entre as várias unidades organizacionais;

b) ferramentas de gerenciamento de projetos: o objetivo dessas ferramentas é disponibilizar aos gerentes de desenvolvimento de software, o rastreamento dos requisitos originais da proposta até a entrega do sistema.
c) ferramentas de apoio: o objetivo dessas ferramentas é complementar o processo de engenharia de software, através de documentação, garantia de qualidade, métricas e controle de versão;
d) ferramentas de análise e projetos: possibilita aos engenheiros de software que criem um modelo do sistema que será construído. Contendo representação do fluxo de controle de dados, conteúdo de dados, representações de processos, especificações de controles e varias outras representações de modelagem. Além de permitir a avaliação da qualidade do modelo;
e) ferramentas de programação: o objetivo dessas ferramentas é apoiar as linguagens de programação no desenvolvimento de sistema. Essa categoria de ferramenta abrange compiladores, editores, depuradores, linguagens de quarta geração, geradores de aplicações, linguagens de consulta e ambientes de programação orientados a objetos;
f) ferramentas de integração e teste: o objetivo dessas ferramentas é adquirir dados a serem usados durante os testes, análise do código-fonte, auxiliar no planejamento, no desenvolvimento, no controle dos testes, na simulação do hardware e outros equipamentos externos;
g) ferramentas de prototipação: o objetivo dessas ferramentas é facilitar/agilizar a criação de protótipos de sistemas;
h) ferramentas de manutenção: o objetivo dessas ferramentas é fazer engenharia reversa do código-fonte para a especificação, analisar a reestruturação do código-fonte e reengenharia de sistemas on-line.

3.4 REQUISITOS

Introduzir uma nova tecnologia em uma organização é um processo difícil, independente da compensação em forma de eficiência e lucros crescentes ou despesas reduzidas que a tecnologia possa oferecer ([FIS1990]).

Segundo [RAF2000], durante o desenvolvimento de software através de ferramenta CASE, é constante a utilização de diagramas para especificar requisitos e critérios do sistema. Desta forma, a ferramenta deve possuir alguns requisitos para facilitar a sua utilização e uma maior aceitação pelos usuários, tais como:

a) inclusão de objetos gráficos: inclusão de objetos no diagrama através do modo *drag_and_drop* (arrastar e soltar). O uso do menu para inclusão de objetos deve ser evitado, podendo aparecer como meio alternativo;
b) texto nos objetos: as ferramentas diagramáticas devem permitir que sejam associados textos aos objetos e ligações de um diagrama. Estes textos devem ficar associados aos objetos correspondentes de maneira perene, e mesmo após operações, como movimentação e redimensionamento dos objetos, a ferramenta não pode perder estas correspondências;
c) operação de *move*: as ferramentas devem permitir que o usuário movimente os objetos pelo diagrama, pois esta é uma operação muito comum quando se está criando ou alterando um diagrama. O cuidado a ser tomado é que as ligações e relações entre objetos e textos associados permaneçam as mesmas, após a operação;
d) operação de *resize*: as operações de redimensionamento de um objeto devem ser feitas através do modo *drag-and-drop*, isto é, que o usuário possa selecionar algum ponto da figura do objeto e arrastá-lo até o tamanho desejado, não comprometendo a figura geométrica;
e) operação de *delete/cut*: eliminar objetos é uma operação bastante comum e simples. O que deve cuidar é para que os dependentes e as ligações ou textos associados ao objeto excluído não fiquem sozinhos ou dependentes;
f) operação de *undo*: a possibilidade de desfazer as operações é também um requisito importante, possibilitando ao projetista retornar ao estado anterior caso cometa algum erro;
g) linhas de grade: as linhas de grade permitem o alinhamento dos objetos no espaço gráfico de um diagrama. Esta é uma operação de caráter estético mas muito útil para a boa qualidade dos diagramas;

h) operação de *merge*: é a possibilidade de unir dois objetos do diagrama em um só, observando, que os dependentes e as ligações originais também fiquem associados ao novo objeto;
i) controle de consistência: as ferramentas devem controlar possíveis erros que possam ocorrer na elaboração do diagrama, avisando o usuário e não permitindo que a operação seja concluída.

3.5 REPOSITÓRIO

Para [PRE1995], o repositório "é um banco de dados que atua como o centro tanto de acúmulo como de armazenagem das informações da engenharia de software". No contexto de ferramentas CASE, o repositório deve ser integrado às ferramentas e manipulado pelos engenheiros de software.

Segundo [MAR1995], o objetivo de um repositório CASE é acumular e armazenar, de forma organizada, uma grande quantidade de conhecimento, complementado em diferentes momentos, por diferentes pessoas, em diferentes lugares.

O repositório é o centro do um ambiente CASE, contendo uma completa representação codificada dos objetos, usados em planejamento, análise, projeto e geração de código, além de disponibilizar recursos para verificar e correlacionar todas essas informações, com o objetivo de garantir a coerência e integridade das mesmas.

Muitos requisitos do repositório CASE são iguais aos de aplicações típicas embutidas nos sistema de gerenciamento de banco de dados comercial. Entre as características padrões dos bancos de dados e os repositórios CASE que suportam o gerenciamento de informações sobre o desenvolvimento de software incluem-se ([PRE1995]):

a) armazenamento de dados não-redundante: o repositório CASE possui um único lugar para o armazenamento de informações pertinentes ao desenvolvimento de sistema, eliminando duplicações perdulárias e potencialmente propensas a erro;
b) acesso de alto nível: facilidade de manusear os dados através de ferramentas CASE;

c) independência de dados: as ferramentas CASE e as aplicações de destino são isoladas da armazenagem física;
d) controle de transações: o repositório gerência interações de múltiplas partes, de uma forma, que mantém a integridade dos dados, quando existem usuários concorrentes e no caso de falha do sistema;
e) segurança: o repositório oferece mecanismos de níveis de permissão e senhas para controlar quem pode ver e modificar as informações contidas nele;
f) consulta a dados e relatórios: o repositório permite acesso direto a seu conteúdo por meio de uma interface com o usuário, além dos relatórios oferecidos como conjunto de ferramentas CASE;
g) abertura: o repositório oferece um mecanismo simples de importação/exportação para possibilitar um carregamento ou transferência de informações;
h) suporte a multiusuários: o repositório permite que vários desenvolvedores trabalhem numa aplicação ao mesmo tempo.

Segundo [PRE1995], o ambiente CASE também faz exigências especiais do repositório que vão além daquilo que está diretamente disponível num banco de dados comercial. Essas exigências podem ser:

a) armazenamento de estrutura de dados sofisticada: o repositório deve permitir o armazenamento de dados simples, e dados complexos, com diagramas, documentos e arquivos. Um repositório também inclui um modelo de informações que descreve a estrutura, os relacionamentos e a semântica dos dados armazenados nele;
b) imposição de integridade: o modelo de informações do repositório também contém normas, políticas, que descrevem normas comerciais válidas e outras imposicões e requisitos sobre a informação a ser introduzida no repositório;
c) interface com a ferramenta rica em semântica: a informação no repositório, deve possuir uma semântica que possibilita a uma variedade de ferramentas interpretar o significado dos dados armazenados;
d) gerenciamento de projeto/processo: o repositório deve conter informações sobre todo o ciclo de desenvolvimento de sistema, possibilitando a coordenação automatizada das atividades de gerenciamento.

A melhor maneira de organizar um repositório CASE é através da utilização de técnicas orientadas a objeto, pois ele visualiza o mundo como uma coleção de objetos que aparecem na tela CASE como caixas, linhas ou outros elementos de diagramas.

Desta forma, como o repositório armazena muitos tipos de objetos, é necessário que ele tenha seu próprio conjunto de classes e métodos. Esses métodos podem ser expressos como coleções de regras ou podem ser algoritmos que processam as informações armazenadas no repositório. Sendo assim, as informações que estão nos diagramas CASE, devem ser coordenadas de forma a garantir que se encaixem de maneira lógica e consistente nas várias regras que o repositório exige ([MAR1995]).

4 SYSTEM ARCHITECT (SA)

4.1 METODOLOGIA

O SA é uma das ferramentas CASE disponíveis no mercado que atende às várias fases do ciclo de desenvolvimento de software, podendo ser classificado nas categorias de:

a) ferramentas que suporta as técnicas de orientação a objetos;
b) ferramentas IE-CASE;
c) ferramentas que suporta as funções de:
   - planejamento de sistemas comerciais;
   - gerenciamento de projetos;
   - ferramenta de apoio;
   - ferramenta de análise e projetos;
   - ferramenta de manutenção.

Ele possui uma flexibilidade de configuração que o torna extremamente abrangente, moldando as informações de acordo com a necessidade da metodologia utilizada na empresa ([CHO1999]).

Entre as metodologias suportadas pela ferramenta, pode-se destacar a *Unified Modeling Language* (UML) para a análise orientada a objetos, permitindo através da ferramenta modelar todos os diagramas suportados por [FUR1998].

Embora a UML não tenha um processo ou método bem definido de como iniciar desenvolvimento da análise de um sistema orientado a objetos, é necessário descrever alguns diagramas e notações (diagrama de casos de uso, diagrama de classes, diagramas de seqüência) que força o analista a modelar certos aspectos do sistema ([POP1999]).

Uma dessas descrições é o diagramas de casos de uso, que tem como objetivo especificar a funcionalidade que o sistema tem a oferecer da perspectiva do usuário, através da representação de atores (usuários) e textos que descreve uma sucessão de passos praticados. Essas informações são adquiridas do usuário através das entrevistas, e facilitam a localização de futuros objetos do sistema ([POP1999] [FUR1998]).

O diagrama de classes, segundo [FUR1998] é a essência da UML resultado de uma combinação de diagramas propostos pela OMT (*Object Modeling Technique*), Booch (método desenvolvido por Grady Booch) e vários outros métodos. Trata-se de uma estrutura lógica estática em uma superfície de duas dimensões mostrando uma coleção de elementos declarativos de modelo, como classes, tipos e seus respectivos conteúdos e relações.

O diagrama de seqüência, segundo [FUR1998] "expõem o aspecto do modelo que enfatiza o comportamento dos objetos em um sistema, incluindo suas operações, interações e histórias de estado em seqüência temporal de mensagem e representação explícita de ativação de operações".

Maiores informações sobre a metodologia UML podem ser adquiridas em [FUR1998].

4.2 ENCICLOPÉDIA

Segundo [POP1999], a enciclopédia do *System Architect* é um banco de dados relacional que contém todos os diagramas e definições, além de alguns arquivos que também fazem parte da enciclopédia constituindo um repositório localizado num único diretório. Esta enciclopédia é criada quando se inicia uma nova modelagem, também permitindo exportar dados de uma enciclopédia já existente.

Desta forma, a enciclopédia é composta ([POP1999]):

a) um banco de dados relacional constituído de duas tabelas e alguns índices;
b) um arquivo para cada diagrama gráfico;
c) um arquivo *metafile* para cada diagrama gráfico;
d) quatro arquivos que determinam a configuração da enciclopédia;
e) um arquivo de *lock* (utilizado para executar em rede).

4.3 REPOSITÓRIO

O repositório principal é composto de duas tabelas (*entity.dbf, relatn.dbf*) e alguns índices para a navegação mais rápida. Essas tabelas são gravadas no formato DBASE III PLUS facilitando o acesso às informações por ser um padrão bem difundido ([SAL1999] [POP1998]).

4.3.1 ESTRUTURA ENTITY.DBF

No geral, cada entrada na tabela "*entity.dbf*"(figura 4), é unicamente identificada por sua *class* + *type* + *name*. Por exemplo, pode-se escrever um relatório numa linguagem de relatório similar ao SQL no *System Architect* que poderia incluir a estrutura da figura 2.

Assumindo que a definição para os dados "*CustomerOrder*" de fato exista, o relatório encontrará apenas uma entrada na tabela "*entity.dbf*", de acordo com os critérios de busca. Pode haver outra entrada com o nome "*CustomerOrder*", mas só pode existir sob alguma outra combinação de *class* e *type*.

FIGURA 2 - SQL DO SYSTEM ARCHITECT

```
SELECT Name
WHERE Class = Definition
WHERE Type = "Data Store"
WHERE Name = CustomerOrder
```

Fonte: [POP1998]

Quando *class* = *diagram* ou *class* = *definition*, como na figura 02, cada combinação de *class* + *type* + *name*, deve ser única. Por outro lado quando a *class* = *symbol* as entradas podem ser ou não ser únicas.

Cada entrada na tabela "*entity.dbf*" deve estar dentro de uma dessas três classes:

a) Diagram (class code = 1);

b) *Symbol (class code = 2)*;

c) *Definition (class code = 3)*;

Cada uma das três classes classificadas acima tem sua própria estrutura de tipos (*type)* apropriado, e cada combinação de classe e tipo possui uma entrada na tabela "*entity.dbf*". Ao total a tabela possui 13 (treze) campos que são utilizados para:

a) NAME : um campo de 31 caracteres;

b) ID : um número de 32 bits, usado como identificador único;

c) CLASS : código da classe, podendo ser diagrama, símbolo ou definição;

d) TYPE : código de tipo que em conjunto com a classe identifica qual é o tipo de informação;
e) NUMBER : é o número do símbolo;
f) TOARROW : é um campo *boolean* que indica a direção do fim de uma linha-símbolo;
g) FROMARROW : é um campo *boolean* que indica a presença de uma cabeça de flecha para o início de uma linha-símbolo;
h) TOASSC : é um código que indica qual a notação gráfica é utilizada na cardinalidade de uma linha-símbolo;
i) FROMASSC : é um código que indica qual a notação gráfica utilizada na cardinalidade de início de uma linha-símbolo;
j) UPDATDATE : é um campo que armazena a data da última atualização;
k) UPDATTIME : é uma campo que armazena a hora da última atualização;
l) AUDIT : campo para armazenar o log do usuário;
m) DESCRIPTN : campo usado para descrição.

4.3.2 ESTRUTURA RELATN.DBF

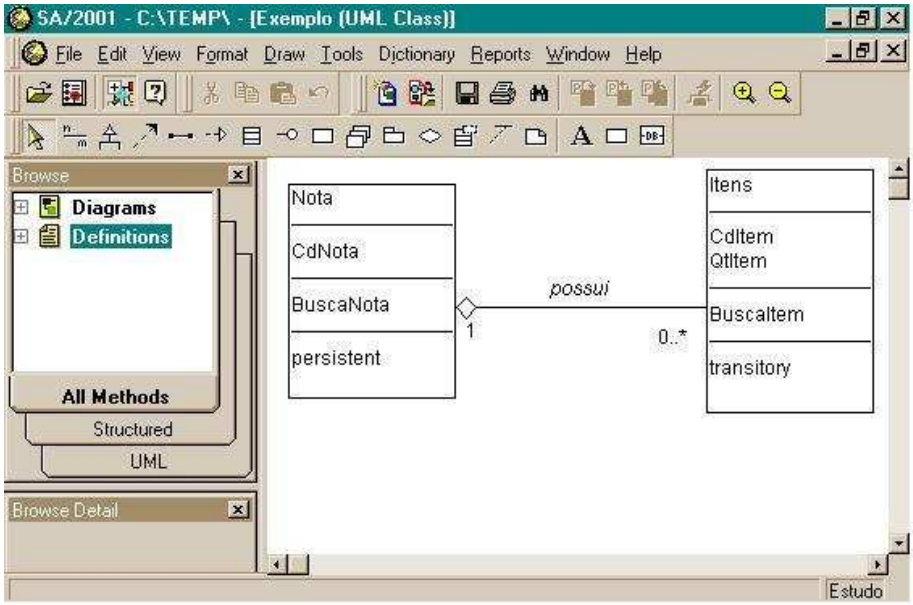
Esta tabela descreve todos os relacionamentos que existem entre os símbolos e as classes existentes nos diagramas que estão gravados na tabela "*entity.dbf*". Isto possibilita existir relacionamento de muitos-para-muitos entre estas classes e símbolos. Cada relacionamento nos gráficos de diagramas geram um par de linhas nesta tabela. Esta tabela possui os seguintes atributos:

a) ID : um número de 32 bits usado como identificador único para a primeira informação (o mesmo ID de *entity.dbf*);
b) ID2 : um número de 32 bits usado como identificador único para a segunda informação (o mesmo ID de *entity.dbf*);
c) RELATION : o tipo de relacionamento entre as duas informações;

4.3.3 EXEMPLO

A figura 03 mostra duas classes que foram modeladas no SA, que serão utilizadas para demonstrar o relacionamento e os dados das tabela *entity.dbf* e *relatn.dbf*.

FIGURA 3 - EXEMPLO DO SA



Ao criar a classe Nota e Itens os registros mostrados na figura 04 foram inseridos na tabela *entity.dbf*. Além dos registros que são criados quando cria uma nova classe, o próprio SA também insere registros ao criar uma enciclopédia.

FIGURA 4 - TABELA ENTITY

Microsoft Access - [ENTITY : Tabela]

Arquivo Editar Exibir Inserir Formatar Registros Ferramentas Janela Ajuda

NAME	ID	CLASS	TYPE	NUMBER	TOARROW	FROMARROW	TOASSC	FROMASSC	UPDATDATE	UPDATTIME	AUDIT	DESCRIPTN
Exemplo	55	1	67	0	0	0	0	0	02/11/00	114541	Estudo	[[[DGX File Name]]]
Nota	56	3	26	0	0	0	0	0	02/11/00	114208	Estudo	[[[Entity State]]]
Itens	57	3	26	0	0	0	0	0	02/11/00	114300	Estudo	[[[Entity State]]]
BuscaNota	58	3	60	0	0	0	0	0	02/11/00	114208	Estudo	[[[Class Name]]]
CdNota	59	3	133	0	0	0	0	0	02/11/00	114208	Estudo	[[[Class Name]]]
CdItem	61	3	133	0	0	0	0	0	02/11/00	114300	Estudo	[[[Class Name]]]
BuscaItem	62	3	60	0	0	0	0	0	02/11/00	114300	Estudo	[[[Class Name]]]
QtItem	63	3	133	0	0	0	0	0	02/11/00	114300	Estudo	[[[Class Name]]]
Itens	64	2	379	0	0	0	0	0	02/11/00	114520	Estudo	[[[Description]]]
Nota	65	2	379	0	0	0	0	0	02/11/00	114520	Estudo	[[[Description]]]
possui	66	2	376	0	0	0	8	9	02/11/00	114520	Estudo	[[[Description]]]
					0	0						

Como pode ser visualizado, as colunas *name* + *class* + *type* determinam o tipo de informação, por exemplo : quando a coluna *class* = 3 e *type* = 26, o conteúdo da coluna *name* será o nome de uma classe no diagrama. Num outro exemplo, quando a coluna *class* = 3 *type* = 133 o conteúdo da coluna *name* será o nome de um atributo. Abaixo são listadas algumas combinações entre as colunas *class* e *type* e os seus significados:

a) classe : *class* = 3 e *type* = 26;
b) atributo : *class* = 3 e *type* = 133;
c) método: *class* = 3 e *type* = 60;
d) herança : *class* = 2 e *type* = 431;
e) associação : *class* = 2 e *type* = 376. O que diferencia o tipo de associação, é o conteúdo das colunas *ToAssc* e *FromAssc*.

Para descobrir a que classe o atributo, o método ou a associação pertence existem duas formas: através da coluna *description* da tabela *entity* (figura 05) ou através da tabela *relatn.dbf* (figura 06).

Através da coluna *description*, existem palavras chaves para a identificação, são elas:

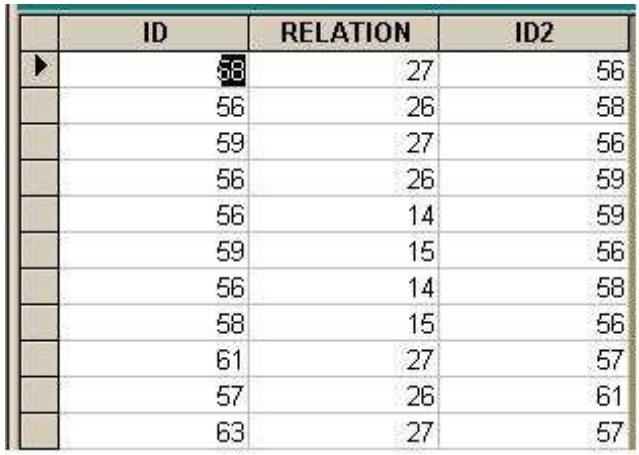
a) [[[To Class]]] : determina a classe de origem numa associação;
b) [[[From Class]]] : determina a classe de destino numa associação;
c) [[[Class Name]]] : determina a classe de um atributo ou método.

FIGURA 5 - ESTRUTURA DA COLUNA DESCRIPTN

	NAME	ID	CLASS	TYPE	DESCRIPTN
	possui	66	2	376	[[[Description]]] [[[Derived]]] F [[[To Member Ordered]]] F [[[From Member Ordered]]] F [[[From Class]]] Itens [[[To Class]]] Nota

Já através da tabela *relatn*, deve-se saber o código (que se encontra na tabela *entity* na coluna *Id*) da "propriedade" (atributo, método ou associação) e o código do tipo de relacionamento (que se encontra na tabela *relatn* na coluna *relation*) que se deseja encontrar.

FIGURA 6 - TABELA RELATN



	ID	RELATION	ID2
▶	58	27	56
	56	26	58
	59	27	56
	56	26	59
	56	14	59
	59	15	56
	56	14	58
	58	15	56
	61	27	57
	57	26	61
	63	27	57

Para a implementação do protótipo foi necessário identificar alguns códigos da tabela *entity*, que são mostrados na tabela 1. Sobre os códigos da tabela *relatn* foi necessário catalogar somente o código 14 da coluna *relation*, utilizado para identificar a classe que o método ou o atributo pertence. As demais informações foram descobertas na coluna *descriptn* da tabela *entity*. Outras informações sobre as estruturas do repositório do SA podem ser encontradas em [POP1998].

TABELA 1 - CÓDIGOS QUE REPRESENTAM OS ELEMENTOS NA MODELAGEM

Class	Type	ToAssc	FromAssc	Significado
3	130	-	-	Indica que é um atributo
3	60	-	-	Indica que é um método
3	26	-	-	Indica que é uma classe
2	431	-	-	Indica uma relação de herança
2	376	8	9	Indica uma relação de agregação
2	376	-	-	Indica uma relação de associação

5 BANCO DE DADOS CACHÉ

A necessidade de possuir um banco de dados é uma parte fundamental para qualquer aplicação que necessite armazenar informações, seja em um arquivo simples ou em um conjunto de arquivos que ocupe centenas de *gigabytes* de espaço. Durante os anos, emergiram vários modelos de banco de dados, sendo o mais difundido o modelo de dados relacional.

O modelo relacional e as ferramentas de SQL estão disponíveis para uma variedade de tarefas, mas infelizmente a falta de desempenho do banco de dados relacional, em aplicações complexas, foi por muitos anos aparente. Além disso, as tabelas utilizadas no modelo relacional, apesar de ser de fácil entendimento, são muito simplistas para representar os dados do mundo real ([INT1999]).

Como uma alternativa para o modelo relacional, os bancos de dados orientados a objetos começaram a aparecer. Estes oferecem um maior nível de sofisticação e permitem aos analistas criar aplicações que refletem relações do mundo real. Infelizmente, a variedade de ferramentas que apoiam o modelo relacional não está disponível para o modelo orientado a objeto. Além disso, o banco de dados orientado a objetos não possui integração com o SQL.

Visualizando a necessidade por um banco de dados de alto desempenho com capacidade de modelar o mundo real e com integração com o SQL, a *InterSystems* criou Caché. O Caché combina o poder da tecnologia de orientação a objeto com o desempenho de uma estrutura de dados multidimensional, permitindo criar aplicações de banco de dados com todas as vantagens do modelo orientado a objeto e relacional ([INT1999]).

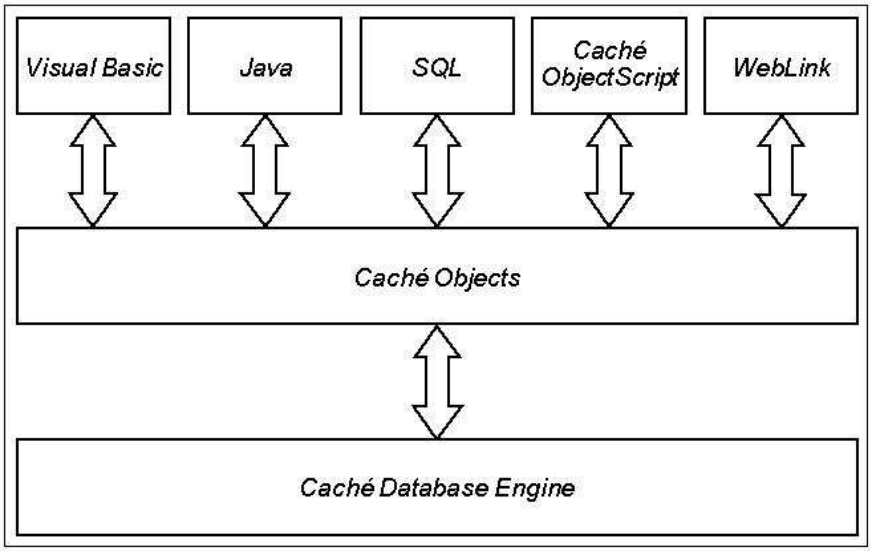
Para usufruir a tecnologia de orientação a objetos, o Caché possui um componente (*Caché Objects*) que serve de intercâmbio entre as ferramentas e a estrutura de dados multidimensional (figura 07). Além do acesso por *Caché ObjectScript*, o Caché expõe seus objetos por ActiveX, Java, WEB, interfaces com SQL, Visual Basic e outras ferramentas.

Segundo [BAE1999], o Caché é um banco de dados que utiliza recursos do modelo de dados multidimensional para romper as limitações de desempenho e modelagem do modelo relacional, oferecendo aos aplicativos desenvolvidos:

a) alta disponibilidade e segurança de dados;

b) eliminação de qualquer armazenamento desnecessário de informações;
c) alto desempenho e escalabilidade na disponibilização de Interfaces Gráficas para Usuários (aplicações GUI, do inglês *Graphical User Interface*) ou *Word Wide Web* (WWW);
d) modificação dos modelos de dados de maneira eficiente;
e) superioridade de recursos em relação ao modelo relacional;
f) um fácil e rápido modelo de aplicações utilizando objetos;
g) uma nova linguagem de programação orientada a objetos : o *Caché Object Script;*
h) alta performance, através dos objetos;
i) rápido desenvolvimento de aplicações;
j) dualização flexível de servidores;
k) desempenho elevado para sistemas Unix, via servidores de processos.

FIGURA 7 - ESTRUTURA DO CACHÉ



Fonte: [INT1999]

5.1 CARACTERÍSTICAS DO CACHÉ

No desenvolvimento para a WEB, o Caché é um ambiente que permite a criação de aplicações de modo simples e rápida, oferecendo desempenho, escalonamento e facilidade de administração das necessidades do mundo dinâmico da Internet, através de tecnologias como ([IPS1999]):

a) *Caché Weblink*: utiliza conexão TCP entre o servidor de WEB e o banco de dados Caché para permitir sessões vinculadas as acesso na Internet;
b) *Caché Weblink Developer*: utilizando software de prateleira para a criação de páginas de WEB, o *Caché Weblink* permite a programação de páginas com *scripts Caché ObjectScript* embutidos. O Developer gera automaticamente páginas no servidor que são executadas quando necessário;
c) *Caché Weblink Event Broker*: consiste em um pequeno *applet* Java que estabelece uma conexão auxiliar à base de dados Caché. Eventos no navegador podem disparar respostas da base de dados se esperar que a página seja submetida;
d) *Caché Weblink Wizard*: gera automaticamente formulários HTML prontos para serem executados pelo Caché.

No desenvolvimento de aplicações para arquitetura cliente/servidor a performance é um ponto crítico que deve ser respeitado. Na tecnologia de servidor de dados, o Caché permite escalar aplicações para atender dezenas de milhares de usuários sem sacrificar a velocidade, através de tecnologias com ([IPS1999]):

a) banco de dados multidimensional: todos os dados são armazenados em vetores multidimensionais esparsos que eliminam a sobrecarga de processamento causada pelos *joins* comuns em bancos de dados relacionais
b) acesso a dados orientado por objetos: os dados podem ser modelados como objetos. O Caché suporta encapsulamento, herança múltipla, polimorfismo, objetos embutidos, referências, coleções e relacionamentos;
c) acesso a dados via SQL: permite o acesso relacional à base de dados Caché através do ODBC e JDBC;
d) acesso a dados multidimensionais: fornece controle direto de estruturas multidimensionais na base de dados Caché;
e) objeto com arquitetura de dados unificada: as classes e tabelas relacionais são automaticamente geradas a partir de um única definição de dados.

No que diz respeito à conectividade e à rapidez de desenvolvimento, as características do servidor de aplicações Caché oferecem aos projetistas toda a flexibilidade e potência necessárias, através de ([IPS1999]):

a) *Caché ObjectScript*: consiste em uma linguagem de programação orientada por objetos para desenvolvimento de *scripts* de lógica empresarial. Suporta simultaneamente o acesso a dados via SQL, multidimensional e orientado por objetos, bem com HTML embutidos;
b) servidores de objetos Caché: os objetos Caché podem ser apresentados como Java, *ActiveX* e C++. Também há suporte à tecnologia *Corba*;
c) Visual Caché e Caché Delphi Link: apresenta elevado desempenho na conectividade com o Visual Basic e o Delphi;
d) *Caché SQL Gateway*: permite que o Caché leia dados em bases relacionais;
e) protocolo Caché distribuídos: armazena dados localmente em servidores e clientes para reduzir o tráfego na rede;
f) mapeamento dinâmico do Namespaces: consiste em um diretório com a localização lógica de dados, objetos e rotinas.

5.2 O MODELO DE DADOS MULTIDIMENSIONAL

Segundo [BAE1999], o modelo de dados multidimensional permite que o analista represente sua estrutura de dados como no mundo real, sem comprometer a estrutura em dimensões restritas em tabelas.

No modelo de dados multidimensional, o enfoque está na coleção de “cubos” e a quantidade de dimensões que cada “cubo” possa ter para atender as reais necessidades da aplicação. Por exemplo, para modelar uma Nota Fiscal em um modelo de dados relacional necessita-se de, pelo menos duas tabelas: uma para a capa e outra para os itens. No modelo multidimensional, pode-se imaginar a nota armazenada em um “cubo” e cada tabela uma dimensão do “cubo”, permitindo incluir quantas dimensões forem necessárias para modelá-la. Isso significa um ganho em performance, visto que os dados, fisicamente e logicamente, encontram-se na mesma estrutura, o que diminui as operações de leitura/gravação em disco ([BAE1999] [RAM1997]).

O Caché organiza os dados do modelo multidimensional em estruturas do tipo árvore, onde cada índice pode indicar o início de uma subárvore e qualquer ramo abaixo de um nó está automaticamente relacionado com ele.

5.3 O CACHÉ OBJECTS

Segundo [BAE1999], o *Caché Objects* é componente da estrutura do banco de dados Caché responsável por disponibilizar, simultaneamente o desempenho e poder de modelagem da estrutura multidimensional de dados do Caché associada às características da tecnologia orientada a objetos.

Dentre as características do *Caché Objects* incluem-se:

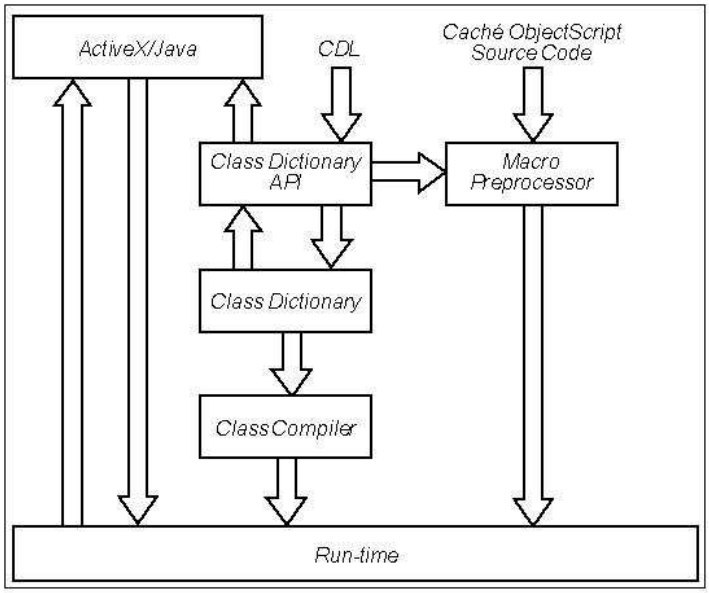
a) fácil e rápido desenvolvimento de aplicações, utilizando objetos;
b) apresentação de uma nova linguagem de programação orientada a objetos o COS;
c) exposição de objetos Caché como objetos nativos ActiveX, especialmente para Visual Basic;
d) exposição de objetos Caché como objetos nativos Java;
e) alto desempenho.

5.3.1 ESTRUTURA DO CACHÉ OBJECTS

O *Caché Objects* é um sistema (figura 08) formado pelos seguintes sub-sistemas ([INT1999]):

a) *Caché Object Architect*: é um ambiente com interface GUI que permite a manipulação completa das classes de *Caché Objects*;
b) *Caché Object Server for ActiveX*: é um componente ActiveX que permite a exposição de objetos Caché como objetos nativos ActiveX para utilização através de ferramentas de desenvolvimento como o Delphi e Visual Basic;
c) *Caché Object Server for Java*: permite a exposição de objetos Caché como objetos nativos Java;
d) *Class Dictionary*: é responsável por armazenar a definições de classes do usuário e do sistema Caché. Cada namespace do Caché possui uma *Class Dictionary*;
e) *Class Dictionary Application Progam Interface* (*Class Dictionary API*): consiste em um conjunto de programas responsáveis pela comunicação entre o *Class Dictionary* e o restante dos componentes do *Caché Objects*;
f) *Class Compiler*: compila as definições de classe armazenadas no *Class Dictionary*.

FIGURA 8 - ESTRUTURA DO CACHÉ OBJECTS



Fonte: [INT1999]

5.3.2 O MODELO DE OBJETOS DO CACHÉ OBJECTS

As operações fundamentais do *Caché Objects* estão baseadas na definição de classes de objetos e subsequente criação, armazenamento, recuperação e manipulação de instâncias específicas dessas classes ([BAE1999]).

Para manter essas operações fundamentais, o *Caché Objects* contém um hierarquia de suporte a objetos (figura 09), os mais difundidos são:

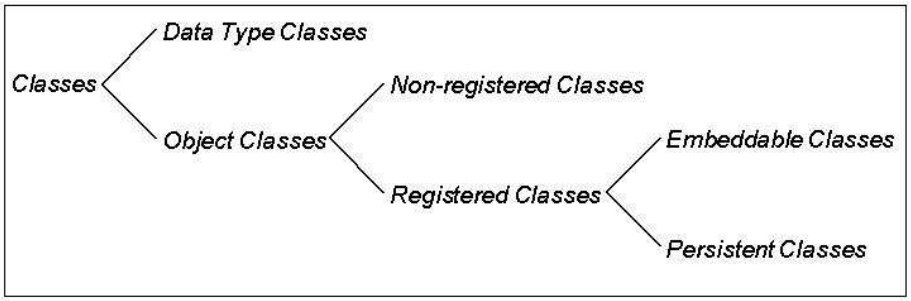
a) Classes Registradas (*Registered Classes*);
b) Classes Persistentes (*Persistent Classes*);
c) Classes Embutidas (*Embeddable Classes*);
d) Classe não Registradas (*Non-Registered Classes*);
e) Classe de Tipos de Dados (*Data Type Classes*).

As Classes de Objetos (*Object Classes* da figura 09) representam entidades específicas do mundo real, e modelam como essas interagem com o mundo externo. Elas dividem-se em Classes Registradas e Classes não Registradas. As Classes Registradas permitem o armazenamento de suas instâncias em disco. As classes não Registradas geralmente especificam classes abstratas e não podem ser instanciadas e/ou armazenadas ([BAE1999]).

As Classes Registradas ainda se subdividem em Classes Persistentes e Classes Embutidas. As Classes Persistentes são objetos que podem ser independentemente armazenados em uma base de dados. As Classes Embutidas estão relacionadas às Classes Persistentes, na forma de atributos destas, não possuem um identificador próprio, e só podem ser manipulados como atributos de Classes Persistentes.

As Classes de Tipos de Dados especificam valores literais, como inteiros, cadeias e datas que são utilizadas para determinar o tipo de dados dos atributos das Classes de Objetos. Elas não possuem definições e métodos e nunca precisam ser instanciadas. Suas instâncias existe implicitamente no sistema Caché. Classes de Tipos de Dados não possui identificador e são identificadas por seus valores.

FIGURA 9 - TIPO DE CLASSES DO CACHÉ OBJECTS



Fonte : [INT1999]

Segundo [BAE1999], o *Caché Objects* oferece amplo suporte a herança, inclusive com herança múltipla. Desta forma, classes de usuários podem ser definidas em função das classes de sistemas já existentes, através do *Caché Object Architect* ou *Caché Class Description Language* (CDL).

5.3.3 CLASSES NO CACHÉ OBJECTS

Segundo [BAE1999], “as classes do *Caché Objects* são formadas por um conjunto de parâmetros/palavras-chave, métodos e propriedades”. As palavras reservadas que permitem manipular as classes do *Caché Objects* encontram-se no anexo 1 ([BAE1999] [INT1999]).

Os parâmetros/palavras-chave de uma classe define toda a parte “comportamental” da classe quando de sua compilação. Através dos parâmetros das classes pode-se definir índices,

atributos-chaves e até mesmo os valores que devem formar a identidade do objeto. Com as palavras-chaves define-se o tipo da classe, sua(s) superclasse(s) e a descrição da mesma ([BAE1999] [INT1999]).

As propriedades ou atributos (como são conhecidos nas linguagens orientadas a objetos) definem o conteúdo de uma classe, seus valores e o estado interno do objeto. As propriedades de uma classe podem ser tipos de dados atômicos (como inteiros e datas), referencias a instâncias de objetos de outras classes (relacionamentos) ou objetos embutidos (objetos que só podem ser acessados na forma de atributos de classes) ([BAE1999] [INT1999]).

Os métodos definem como a classe se comunica com o meio externo. Existem três tipos de métodos no modelo de objetos do *Caché Objects* ([BAE1999]) :

a) métodos de instância: são métodos que só podem ser executados através de um objeto instanciado, agindo sobre a instância do objeto que o invocou;
b) métodos de classe: são métodos executados sem a presença de um objeto instanciado da classe, geralmente utilizados para instanciar objetos;
c) métodos de consulta: também conhecidos como *Query* no sistema Caché, agem sobre todo o conjunto de instâncias em disco dos objetos. Sendo utilizados para processos de recuperação, análise e consulta de instâncias de objetos.

5.3.4 CACHÉ CLASS DEFINITION LANGUAGE (CDL)

Segundo [INT1999], o *Class Definition Language* (CDL) é uma linguagem do Caché utilizada para definição de classes, que pode ser editada em um editor de texto no formato ASCII. Este arquivo, para ser “entendido” pelo sistema Caché, deve ser carregado e compilado pelos programas do *Class Dictionary API*.

Os arquivos CDL também são úteis quando se deseja mover ou copiar classes de um sistema Caché para outro. As classes são simplesmente exportadas para arquivos CDL, importadas para o novo sistema e compiladas. Um arquivo CDL pode conter várias definições de classe ([BAE1999]).

No anexo 1 estão descritas as principais características de sintaxe da linguagem CDL.

6 COMPILADORES

Neste capítulo é analisada a semelhança entre os passos executados por um compilador com as passos do protótipo construído, pois ambos possuem como objetivo transformar uma linguagem fonte, que tem uma determinada estrutura, para uma linguagem destino com uma estrutura diferente. A seguir são apresentadas as fases de análise léxica, análise sintática, análise semântica e geração de código que existe nos compiladores para a transformação de uma linguagem fonte para linguagem destino e uma análise da semelhança com as fases do protótipo.

6.1 ANÁLISE LÉXICA

O analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma seqüência de *tokens*, que são seqüência de caracteres tendo um significado coletivo, que será utilizada pela análise sintática ([AHO1995]).

O analisador léxico executa usualmente uma série de funções, não obrigatoriamente ligadas à análise léxica propriamente dita, porém todas de grande importância, algumas delas são ([JOS1987]) :

a) eliminação de delimitadores e comentários;
b) conversão numérica;
c) identificação de palavras reservadas;
d) recuperação de erros.

6.2 ANÁLISE SINTÁTICA

A análise sintática é a segunda fase de um compilador Ela cuida exclusivamente da forma das sentenças da linguagem, através da recepção de uma seqüência de *tokens* extraídos pelo analisador léxico. A partir desta seqüência, o analisador sintático efetua uma verificação acerca da ordem de apresentação dos *tokens*, identificando em cada situação o tipo da construção sintática por eles formada ([AHO1995] [JOS1987]).

A análise sintática engloba diversas funções, algumas delas são ([JOS1987]):

a) detecção de erros de sintaxe;

b) recuperação de erros;
c) correção de erros.

6.3 ANÁLISE SEMÂNTICA E GERAÇÃO DE CÓDIGO

A terceira fase de um compilador refere-se à tradução propriamente dita do programa-fonte para a forma do código-objeto. Em geral, a geração de código vem acompanhada de muitas implementações das atividades de análise semântica, responsáveis pela captação do sentido do texto-fonte que é uma operação essencial à realização da tradução do mesmo. Algumas das atividades da análise semântica são ([JOS1987]):

a) criação e manutenção de tabelas de símbolos;
b) associar os símbolos os correspondentes atributos;
c) representar tipos de dados;
d) efetuar a tradução do programa;
e) geração de código.

6.4 COMPARAÇÃO DAS FASES DOS COMPILADORES

O objetivo dos compiladores, de uma maneira ampla, é pegar uma linguagem perceptível ao ser humano e transformá-la para que seja “entendida” pelo computador, ou seja, realizar uma tradução ou transformação a partir de uma linguagem fonte para outra linguagem destino.

A execução do protótipo construído neste trabalho possui uma semelhança com um compilador, pois ele está analisando uma “linguagem” que está representada em forma de tabelas no repositório da ferramenta CASE e a transformando em um código-fonte CDL em formato ASCII para o banco de dados Caché.

A fase léxica dos compiladores pode ser comparada com as atividades de leitura do repositório e montagem da estrutura de classes em memória, pois ambas as atividades são realizadas sobre informações relevantes, conversão dos tipos de dados e a identificação de elementos chaves (palavras chaves).

A fase sintática, com o objetivo de analisar possíveis erros ou correção, não está implementada no protótipo, pois não é verificada consistência entre os elementos. Espera-se que esta consistência exista em razão do uso da ferramenta CASE.

Indo para a fase semântica, que corresponde à geração de código-objeto, o protótipo realiza a geração do código CDL, que corresponde na união das palavras reservadas da linguagem CDL com as informações adquiridas na leitura do repositório .

Os passos do protótipo estão melhor descritos no capítulo 7.

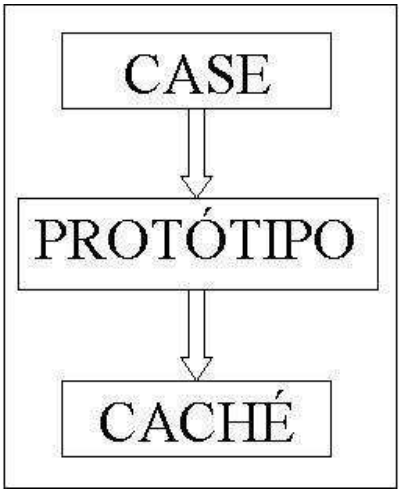
7 PROTÓTIPO

Neste capítulo é apresentado o protótipo de software de geração de código *Class Definition Language* (CDL) para banco de dados Caché a partir do repositório da ferramenta CASE *System Architect* (SA), juntamente com sua especificação.

7.1 INTRODUÇÃO

Segundo [MAR1995], as técnicas de orientação a objetos e a tecnologia CASE convivem naturalmente e formam um poderoso mecanismo para desenvolvimento de software. Com base nestas informações foi desenvolvido o protótipo, que tem como objetivo ler as informações do repositório da ferramenta CASE SA e a partir dessa aquisição, gerar o código que pode ser compilado no banco de dados Caché.

FIGURA 10 - OBJETIVO DO PROTÓTIPO



Para o desenvolvimento deste protótipo, utilizou-se o ambiente de programação Delphi 5.0, onde foram implementados seis classes e um formulário responsáveis pela leitura e geração de código. A ferramenta CASE *System Architect* deve estar configurada para utilizar a notação UML, que será empregada para modelar o banco que se deseja gerar o código. O banco de dados Caché será utilizado para compilar o código gerado pelo protótipo.

7.2 ESPECIFICAÇÃO DO PROTÓTIPO

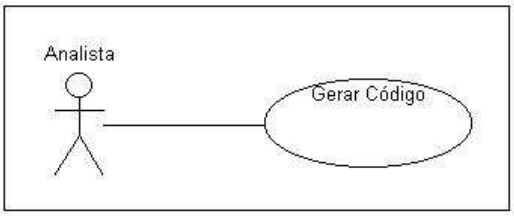
Para a especificação do protótipo foi utilizado a UML, que é apresentado através do diagrama de caso de uso, do diagrama de classe e do diagrama de seqüência. Estes diagramas serão apresentados a seguir e foram construídos no próprio SA.

7.2.1 DIAGRAMA DE CASO DE USO

Neste protótipo foi observado apenas um caso de uso (figura 11), que se refere à interação do analista para a geração de código.

O analista (figura 11) é responsável por configurar o diretório onde se encontra a base de dados do repositório. Após está configuração ele pode iniciar o processo de geração de código. O processo de geração de código está dividido em três tarefas: leitura da base de dados, geração de código e por último salvar o código que foi gerado caso for de interesse do analista.

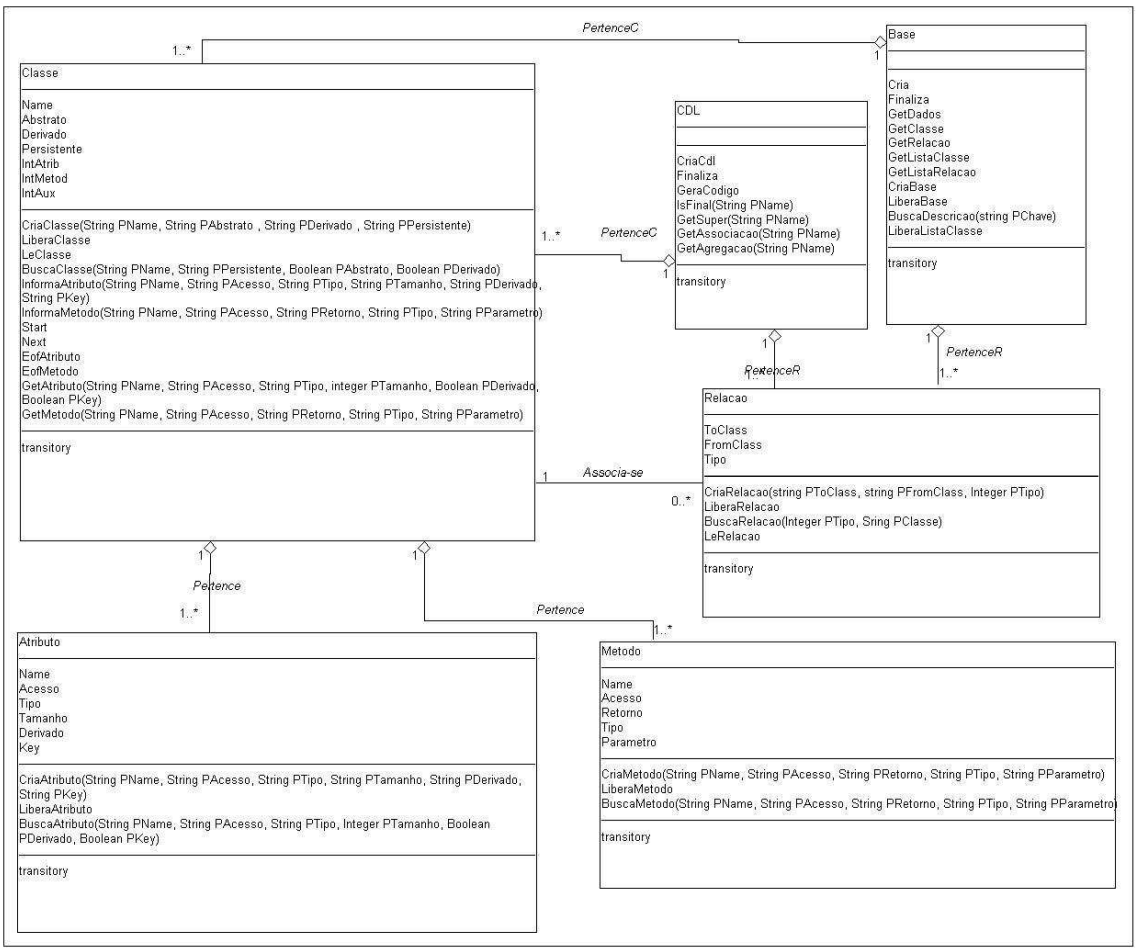
FIGURA 11 - DIAGRAMA DE CASO DE USO



7.2.2 DIAGRAMA DE CLASSE

No desenvolvimento do protótipo foram identificadas seis classes (figura 12) que são utilizadas para armazenar as informações do repositório da ferramenta CASE e posteriormente gerar o código em CDL.

FIGURA 12 - DIAGRAMA DE CLASSE



A classe Base, apresentada na figura 12, é responsável por ler as informações do repositório da ferramenta SA e criar as classes Relacao e Classe. Além das agregações, ela não contém atributos, somente os métodos que são:

a) cria (): operação responsável por criar a classe Base;
b) finaliza (): libera o espaço em memória que a classe está utilizando;
c) getDados (): operação responsável por ler as informações do banco de dados;
d) getClasse (): função que retorna o nome das classes criadas no SA;
e) getRelacao (): função que retorna as associações criadas no SA;
f) getListaClasse (): função que retorna uma lista das classes Classe criadas ao executar o protótipo;
g) getListaRelacao (): função que retorna uma lista das classes Relacao criadas ao executar o protótipo;

h) criaBase (): operação privada responsável por criar os componentes de acesso a base de dados;
i) liberaBase (): operação privada responsável por destruir os componentes de acesso a base de dados;
j) buscaDescricao (PChave): função privada que retorna o nome da classe, atributo ou método que se encontra na coluna *descriptn* da tabela *entity.dbf*.
k) liberaListaClasse (): operação privada que libera o espaço em memória das classes que estão agregadas (Classe e Relacao).

A classe Cdl, apresentada na figura 12, é responsável por gerar o código em CDL, através da união das palavras reservadas (anexo 1 e anexo 3) com as informações adquiridas através da leitura do repositório feita pela classe Base, possuindo os seguintes métodos:

a) criaCdl (): operação responsável por criar a classe Cdl;
b) finaliza (): libera o espaço em memória que a classe está utilizando;
c) geraCodigo (): operação responsável por gerar o código em CDL;
d) isFinal (PName) : função privada que indica se a classe possui filhas, retornando verdadeiro ou falso;
e) getSuper (PName) : função privada que retorna o nome da classe superior;
f) getAssociacao (PName) : função privada que retorna o nome da classe que está associada à classe passada pelo parâmetro;
g) getAgregacao (PName) : função privada que retorna o nome da classe que está agregada à classe passada pelo parâmetro.

A classe Classe, apresentada na figura 12, contém as informações das classes que são modeladas no diagrama de classe da ferramenta CASE SA. Ela contém os seguintes atributos:

a) name: é um campo *String*, que possui o nome da classe;
b) abstrato: é um campo *Boolean*, que indica se a classe é abstrata;
c) derivado: é um campo *Boolean*, que indica se a classe é derivada;
d) persistente: é um campo *String*, que indica se a classe é persistente;
e) intatrib: é um campo *Integer*, que indica a quantidade de atributos da classe;
f) intmetod: é um campo *Integer*, que indica a quantidade de métodos da classe;

g) intaux: é um campo *Integer*, utilizado para auxiliar na leitura dos atributos e métodos;

Existem ainda os métodos da classe que são :

a) criaClasse (PName, PAbstrato, PDerivado, Ppersistente): operação responsável por criar a classe com os dados que são passados como parâmetros;
b) liberaClasse (): libera o espaço em memória que a classe está utilizando;
c) leClasse () : retorna o nome de uma da classe;
d) buscaClasse (PName, Ppersistente, PAbstrato, PDerivado) : busca as informações da classe através dos parâmetros;
e) informaAtributo (PName, PAcesso, PTipo, PTamanho, PDerivado, PKey) : informa os dados que são passados como parâmetros, para a criação dos atributos;
f) informaMetodo (PName, PAcesso, PRetorno, PTipo, Pparametro) : informa os dados que são passados como parâmetros, para a criação dos métodos;
g) start() : inicia o campo intaux para a leitura dos atributos e métodos;
h) next() : incrementa o campo intaux;
i) eofAtributo() : verifica se é o final dos atributos;
j) eofMetodo() : verifica se é o final dos métodos;
k) getAtributo(PName, PAcesso, Ptipo, PTamanho ,PDerivado, PKey): busca as informações dos atributos da classe através dos parâmetros;
l) getMetodo(PName, PAcesso, PRetorno, PTipo, Pparametro) : busca as informações dos métodos da classe através dos parâmetros;

A classe Atributo, apresentada na figura 12, contém as informações dos atributos das classes que são modeladas no diagrama de classe do CASE SA. Ela contém os seguintes atributos:

a) name: é um campo *String*, que possui o nome do atributo;
b) acesso: é um campo *String*, que indica se o atributo é *Public* ou *Private*;
c) tipo: é um campo *String*, que indica o tipo do atributo;
d) tamanho: é um campo *integer*, que indica o tamanho de um atributo quando ele for *string*;
e) derivado: é um campo *Boolean*, que indica se o atributo é derivado;

f) key: é um campo *Boolean*, que indica se o atributo é chave.

Já os métodos dessa classe são:

a) criaAtributo(PName, PAcesso, PTipo, PTamanho, PDerivado, PKey) : operação responsável por criar a classe Atributo com os dados passados pelos parâmetros;
b) liberaAtributo() : libera o espaço em memória que a classe está utilizando;
c) buscaAtributo(PName, PAcesso, Ptipo, Ptamanho, PDerivado, PKey) : busca as informações da classe através dos parâmetros.

A classe Método, apresentada na figura 12, contém as informações dos métodos das classes que são modeladas no diagrama de classe da ferramenta CASE SA. Ela contém os seguintes atributos:

a) name: é um campo *String*, que contém o nome do método;
b) acesso: é um campo *String*, que indica se o método é *Public* ou *Private*
c) retorno: é um campo *String*, que indica o tipo de retorno caso o método seja uma *Function*;
d) tipo: é um campo *String*, que indica se o método é uma *Function* ou *Procedure*;
e) parametro: é um campo *String*, que contém os parâmetros do método.

Já os métodos dessa classe são:

a) criaMetodo (PName, PAcesso, PRetorno, PTipo, PParametro): operação responsável por criar a classe Metodo com os dados passados pelos parâmetros;
b) liberaMetodo(): libera o espaço em memória que a classe está utilizando;
c) buscaMetodo(PName, PAcesso, PRetorno, PTipo, PParametro): busca as informações da classe através dos parâmetros;

A classe Relação, apresentada na figura 12, contém as informações dos três tipos de associações que o protótipo reconhece. Os atributos dessa classe são:

a) toclass : é um campo *String*, que indica o nome da classe da origem da associação;
b) fromclass : é um campo *String*, que indica o nome da classe de destino da associação;
c) tipo : é um campo *Integer*, que indica o tipo de associação entre as classes (1 = herança; 2 = associação; 3 = agregação).

Os métodos dessa classe são:

a) criaRelacao (PToClass, PfromClass,. PTipo) : operação responsável por criar a classe Relacao com os dados passados por parâmetro;
b) liberaRelacao () : libera o espaço em memória que a classe está utilizando;
c) leRelacao () : retorna o nome do tipo das associações (1 = herança; 2 = associação; 3 = agregação);
d) buscaRelacao (Ptipo, PClasse): retorna o nome da classe, conforme as condições passadas pelos parâmetros.

7.2.3 DIAGRAMA DE SEQÜÊNCIA

Na elaboração do protótipo identificou-se que o diagrama de seqüência possui duas fases. A fase 01, apresentada na figura 13, identifica a seqüência de passos que são realizados para a leitura das informações do repositório da ferramenta CASE SA.

A fase 02 apresentada na figura 14, identifica a seqüência de passos que são realizados para a geração de código em CDL.

FIGURA 13 - DIAGRAMA DE SEQÜÊNCIA FASE 01

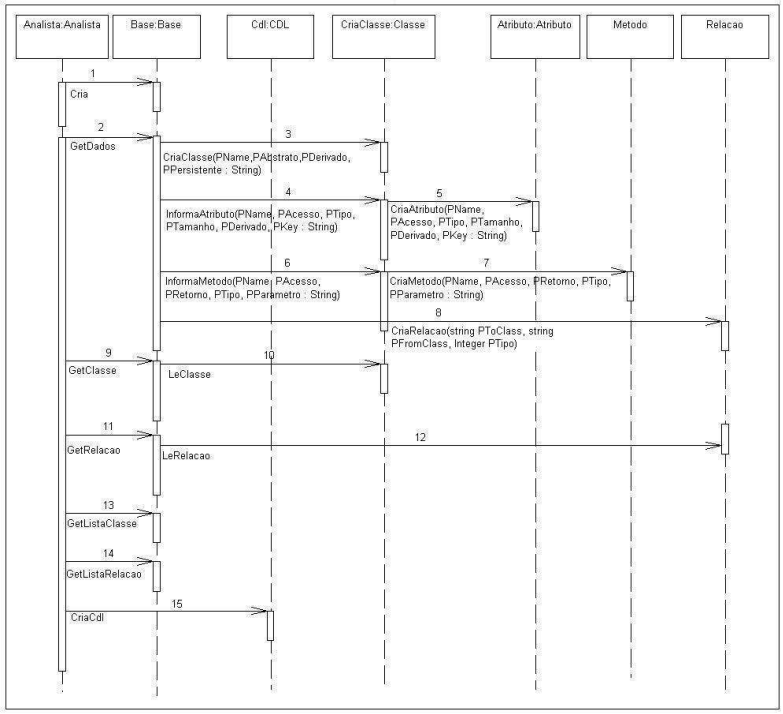
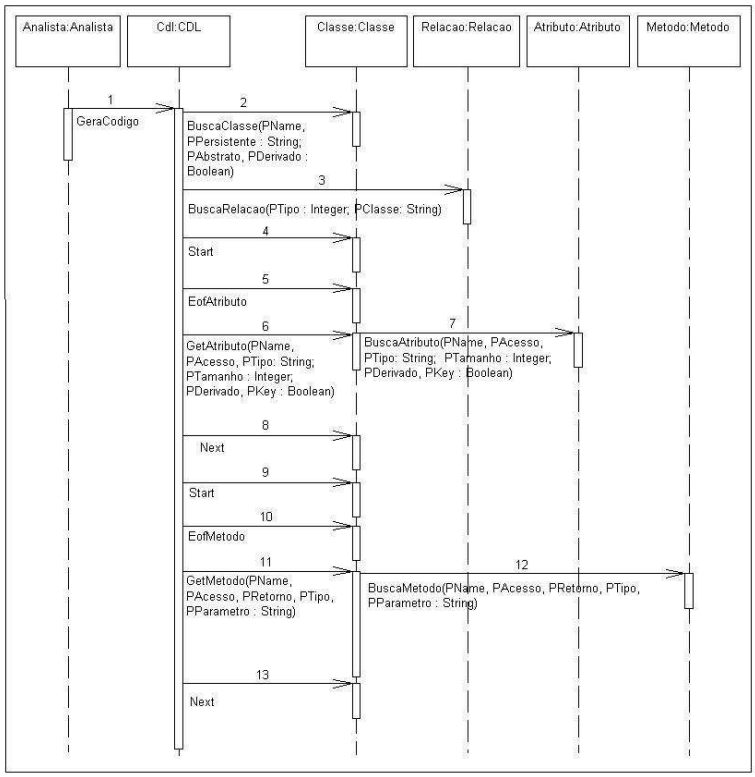


FIGURA 14 - DIAGRAMA DE SEQÜÊNCIA FASE 02



7.3 IMPLEMENTAÇÃO DO PROTÓTIPO

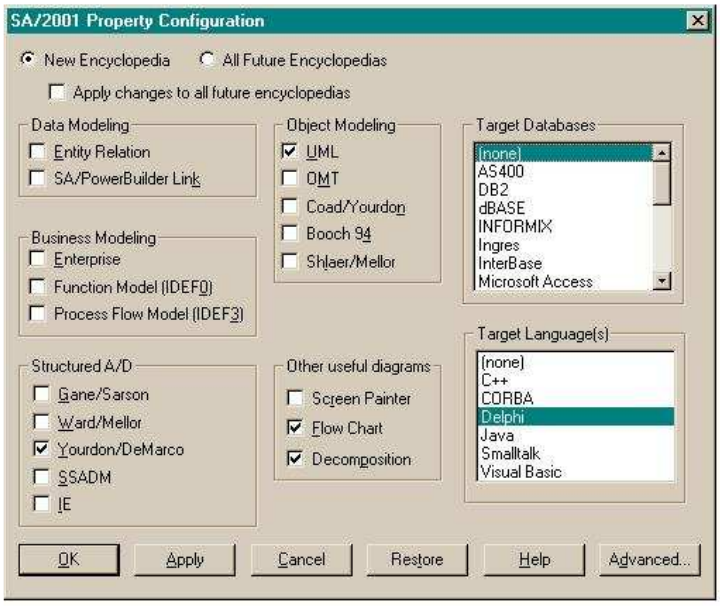
O protótipo foi implementado no ambiente de programação Delphi 5.0, onde foram empregados os conceitos de orientação a objetos para desenvolver seis classes responsáveis pela leitura e geração de código. Estas por sua vez, podem ser acrescentadas em qualquer sistema que se deseje obter a leitura do repositório do SA com geração de código em CDL.

No anexo 2 encontra-se a definição dessas seis classes. As classes Cdl e Base são apresentadas com os detalhes de sua implementação.

7.4 CONFIGURAÇÃO DA FERRAMENTA CASE

A ferramenta CASE que o protótipo utiliza para obter as informações para a geração de código é o *System Architect* 2001 versão 6.7.3. Para ser utilizada pelo protótipo, a ferramenta deve ser configurada conforme figura 15. O usuário possui duas opções para configurar a ferramenta. A primeira é quando se cria uma nova enciclopédia e a segunda é através do menu *Tools\ Customize Method Support\ Encyclopedia Configuration*.

FIGURA 15 - CONFIGURAÇÃO DO SA



Apesar das várias opções que o SA oferece para criar atributos e métodos, apenas algumas são utilizadas para a geração de código em CDL.

Para os atributos, são:

a) *name*: nome do atributo;
b) *visibility*: informa se o atributo é público ou privado;
c) *string length*: indica o tamanho de uma campo *string*;
d) *set type*: indica o tipo do campo;
e) *Derived* : indica se o campo é derivado;
f) pk: indica se o campo é *primary key* .

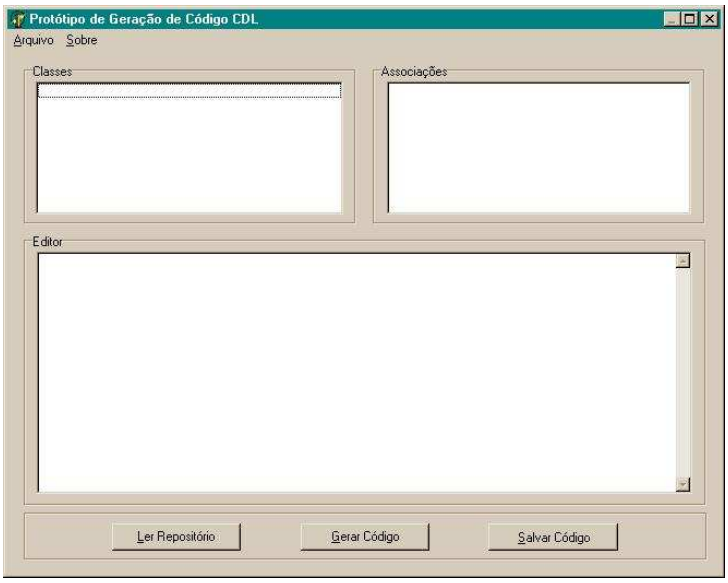
Para os métodos, são:

a) *name*: nome do método;
b) *format parameters*: parâmetros do método;
c) *visibility*: informa se método é público ou privado;
d) *method type*: tipo do método;
e) *return type*: retorno do tipo do método.

7.5 EXECUTANDO O PROTÓTIPO

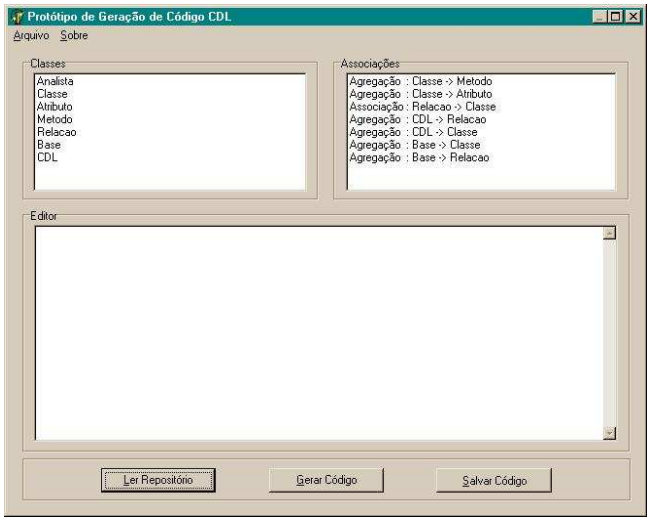
Ao executar o protótipo será mostrada uma tela (figura 16), onde o usuário terá todas as opções disponíveis para a geração de código em CDL. Estas opções (Ler Repositório, Gerar Código e Salvar Código) podem ser acessadas através do menu Arquivo ou através dos botões que se encontram na parte inferior da tela.

FIGURA 16 - TELA DO PROTÓTIPO



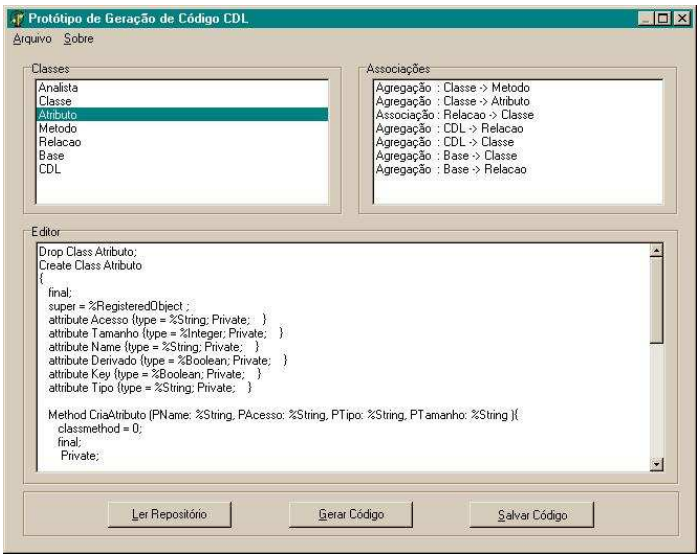
A opção de Ler Repositório deve ser a primeira a ser executada, pois é ela a responsável por ler as informações das tabelas *entity.dbf* e *relatn.dbf* do repositório do SA. Ao executar esta opção os *frames* Classes e Associações (figura 17), serão carregados com as informações das classes e associações do modelo que o *Alias* "TCC_SA" (indica o local das tabelas *entity.dbf* e *relatn.dbf*) estiver configurado. O exemplo da figura 18 é a leitura do repositório da modelagem do próprio protótipo (figura 12).

FIGURA 17 - LEITURA DO REPOSITÓRIO



A opção de Gerar Código é a responsável por gerar o código em CDL. Para ela ser executada deve-se selecionar a classe ou classes no *frame* Classes que se deseja obter o código , o qual será mostrado no *frame* Editor. O exemplo da figura 18 mostra a geração de código da classe Atributo.

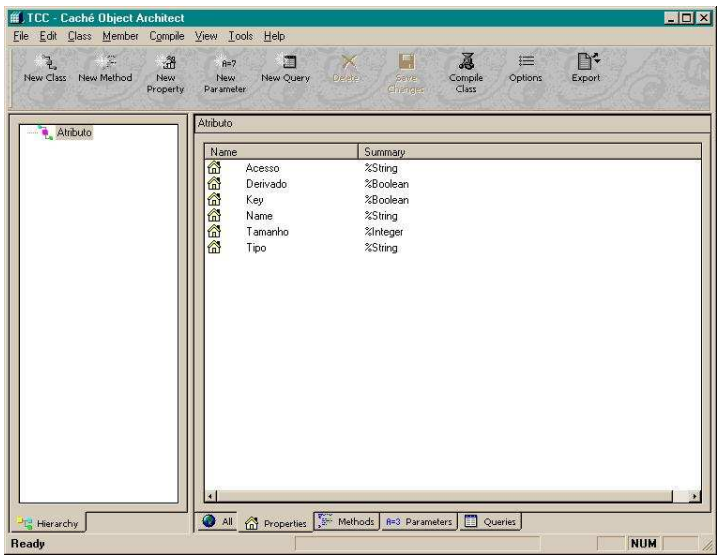
FIGURA 18 - GERACÃO DE CÓDIGO



Na opção Salvar Código salva o conteúdo do *frame* Editor em formato .cdl no diretório em que o usuário especificar, terminando o processo de geração de código por parte do protótipo.

Para compilar o código gerado pelo protótipo no Caché, deve-se escolher a opção *Object Architect* (figura 19) que permite através da opção de menu *File/Import/Import CDL* escolher o arquivo em formato .cdl que se deseja compilar.

FIGURA 19 - COMPILANDO O CÓDIGO



7.6 LIMITAÇÃO DO PROTÓTIPO

No desenvolvimento do protótipo ficou limitado somente a implementação de três recursos primordiais da orientação a objetos, que se refere a herança, associação e agregação.

Também não se encontra no protótipo, uma forma flexível de em tempo de execução determinar o diretório para o *Alias* “TCC_SA” do repositório do SA que se deseja gerar o código.

7.7 ESTUDO DE CASO

Como exemplo de estudo de caso, foi elaborado um sistema fictício que tem como objetivo mostrar a execução do protótipo e principalmente o código fonte em CDL (anexo 3).

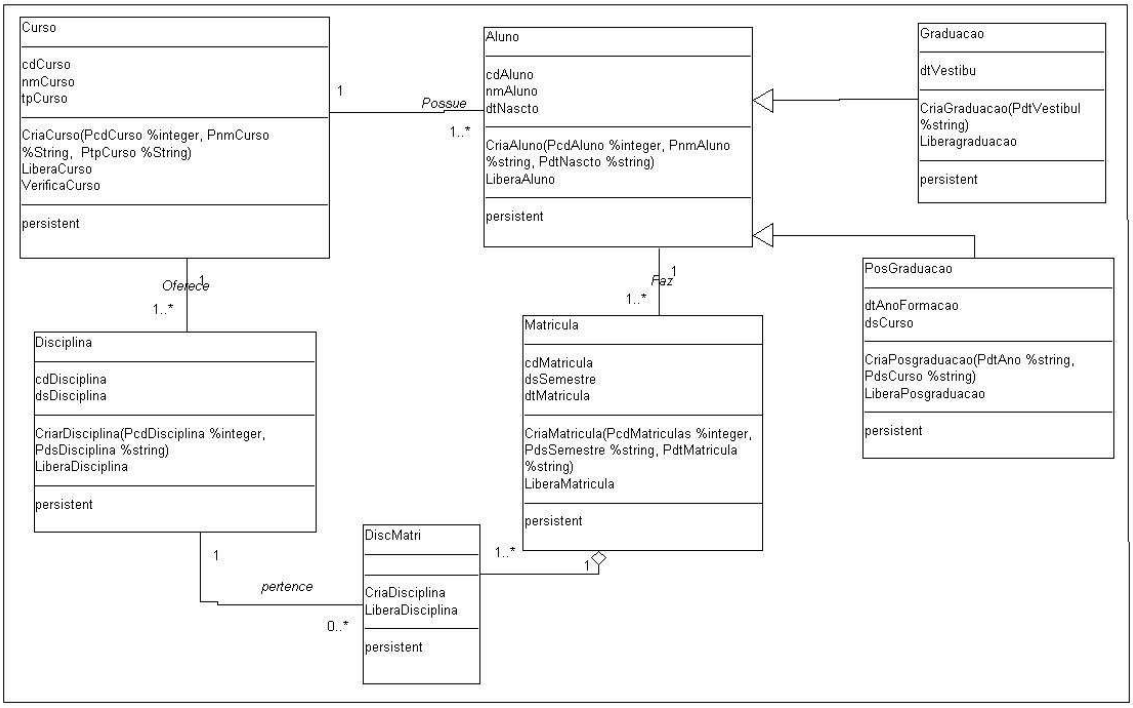
O sistema fictício consiste em controlar as matrículas de alunos de graduação e de pós-graduação de uma universidade. Os alunos de graduação possuem os seguintes dados de cadastro: código do aluno, nome, data de nascimento, curso e data em que foi aprovado no

vestibular. Já os alunos de pós-graduação possuem código, nome, data de nascimento, curso que está sendo freqüentado, ano e curso de formação.

O aluno, no início de cada semestre, precisa fazer sua matricula, optando pelas disciplinas que vai cursar naquele semestre. A matrícula é composta do código do aluno, número do semestre e data da matrícula.

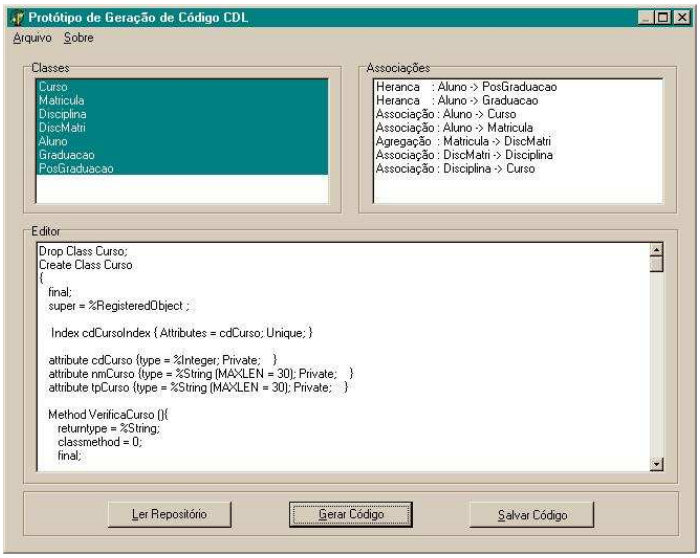
Na figura 20 é mostrado o diagrama de classes modelado no SA através da especificação UML.

FIGURA 20 - DIAGRAMA DE CLASSE DO ESTUDO DE CASO



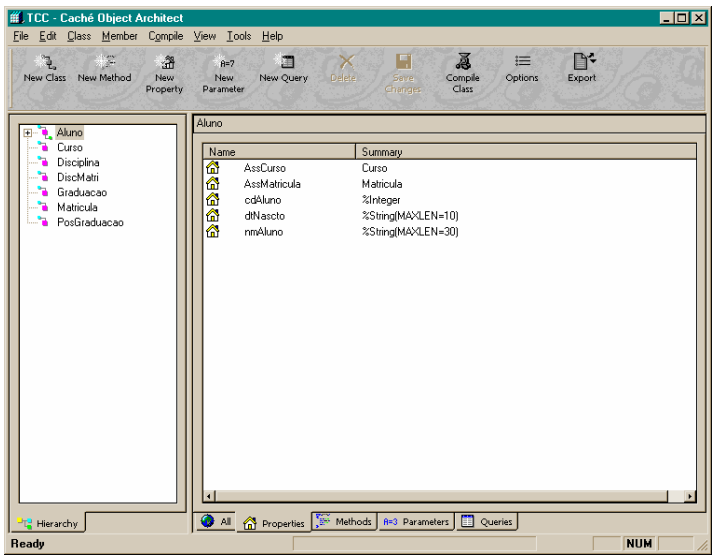
Na figura 21 é mostrado o protótipo já carregado com todas as definições que foram apresentadas na figura 20.

FIGURA 21 - DEFINIÇÃO DAS CLASSES NO PROTÓTIPO



Na figura 22, são mostradas as classes que foram criadas no Caché através do código que se encontra no anexo 3.

FIGURA 22 - CLASSE CRIADAS NO CACHÉ



8 CONCLUSÃO

Através deste trabalho foi possível aprimorar o conhecimento sobre as tecnologias de banco de dados orientado a objetos e ferramenta CASE. Também verificou-se através do levantamento bibliográfico, que a união dessas duas tecnologias formam um poderoso mecanismo para desenvolvimento de sistemas complexos, que necessitam cada vez mais modelar os sistemas de acordo com o mundo real.

Sobre a ferramenta CASE *System Architect* verificou-se que ela atende às várias fases do ciclo de desenvolvimento de software, possuindo uma flexibilidade de configuração que a torna extremamente abrangente, permitindo modelar as informações de acordo com a necessidade da metodologia utilizada. Entre as várias metodologias suportadas pela ferramenta, pode-se destacar a UML a qual foi utilizada para análise do protótipo e para análise do banco que se deseja gerar o código em CDL.

Verificou-se também que a enciclopédia do *System Architect* é um banco de dados relacional que contém todos os diagramas e definições em duas tabelas com formato DBF, além de alguns arquivos que também fazem parte da enciclopédia constituindo um repositório localizado num único diretório e de fácil acesso.

Sobre o banco de dados Caché, verificou-se que ele combina o poder da tecnologia de orientação a objeto com o desempenho de uma estrutura de dados multidimensional, permitindo criar aplicações de banco de dados com todas as vantagens do modelo orientado a objeto. Também foi verificado que o CDL é uma linguagem do banco de dados Caché muito útil para a definição de classes, que pode ser editada em um editor de texto no formato ASCII o que permitiu que fosse desenvolvido o protótipo.

Finalmente sobre o desenvolvimento do protótipo, conseguiu-se alcançar os objetivos desejados através da implementação de seis classes e um formulário em ambiente de programação Delphi 5, que são utilizadas para a geração de código em CDL para banco de dados Caché através do repositório da ferramenta CASE SA. Mas encontrou-se uma carência referente a documentação do significado dos códigos da tabela *entity.dbf*, onde foi necessário analisar e descobrir os valores dos códigos que representam as classes, atributos, métodos e associações relevantes para a implementação do protótipo.

8.1 SUGESTÕES

Como sugestões para continuação deste trabalho inclui-se:

a) estudar a definição de outra linguagem que possua conceitos de orientação a objetos e implementar uma nova classe para geração de código utilizando-se das classes de leitura do repositório do SA, já implementadas no protótipo;
b) implementar novas classes para a leitura de todos os conceitos de orientação a objetos que o SA disponibiliza em sua ferramenta;
c) estudar outras metodologias suportadas pela ferramenta SA e implementar novas classes de leitura do repositório com base nas já existentes neste trabalho.

ANEXO 1 - GUIA DE REFERENCIA CDL

Nos tópicos a seguir estão descritos as principais características para a criação de classes através da linguagem CDL.:

A1.1 COMENTÁRIOS

Comentários não podem aparecer dentro de alguma parte do arquivo CDL que é formado por código Caché ObjectScript (COS).

Comentário de Linha : / /

Comentários em Geral: abertura com /* fechamento com *\

A1.2 DELIMITADOR DE BLOCO DE COMANDOS

Abertura com { fechamento com }

A1.3 COMANDO CREATE CLASS

Sintaxe: CREATE CLASS **classname** {classkeyword1; classkeyword2; ...}

O comando CREATE CLASS cria uma nova definição de classe no *Class Dictionary*. Se a classe que esta sendo definida já existe, um erro ocorre. **Classname** é uma palavra alfanumérica, que começa com a letra ou sinal de percentual (%), e define o nome da classe que esta sendo criada. **Classkeywordn** correspondem a palavras reservadas para definição de classes e são explicadas a seguir.

A1.4 COMANDO DROP CLASS

Sintaxe: DROP CLASS **classname**

O comando DROP CLASS apaga um definição de classe no Class Dictionary. **Classname** define o nome da classes à ser apagada.

A1.5 ABSTRACT

Sintaxe: ABSTRACT;

Define que a classe é abstrata.

A1.6 FINAL

Sintaxe: FINAL;

Define que a classes é final. Não pode ter subclasses.

A1.7 PERSISTENT

Sintaxe: PERSISTENT;

Define que a classe é persistente.

A1.8 SUPER

Sintaxe: SUPER = **classename1, classname2, ..., classnamen**;

Define a(s) superclasse(s) da classe.

A1.9 SYSTEM

Sintaxe: SYSTEM;

Define que a classe é uma classe de sistema.

A1.10 DESCRIPTION

Sintaxe: DESCRIPTION = **descrição**

Define um descrição para a classe.

A1.11 PARAMETER

Sintaxe: PARAMETER **ParameterName {ParameterDefinition}**

Define os parâmetros da classe.

ParameterName define o nome do parâmetro.

ParameterDefinition possui uma sintaxe similar às definições de classe e possui apenas uma palavra reservada:

a) **DEFAULT:** define o nome do atributo cujo o valor é padrão para esse parâmetro ou o valor do parâmetro propriamente dito.

Sintaxe : DEFAULT = **AttributeName** onde:

- **AttributeName:** é o nome do atributo de valor padrão ou um valor propriamente dito (um *String* ou *Integer*, por exemplo).

A1.12 ATTRIBUTE

Sintaxe: ATTRIBUTE **AtributeName {AtributeDefinition}**

Define as propriedades/atributos da classe.

AtributeName define o nome do atributo/propriedade.

AtributeDefinition define as características do atributo. Também possui uma sintaxe muito similar às definições de classe e pode receber (entre várias) as seguintes palavras reservadas:

a) **CALCULATED:** define que este é um atributo calculado.

Sintaxe: CALCULATED;

b) **FINAL:** define que o atributo é final e não pode ser sobreposto nas subclasses.

Sintaxe: FINAL;

c) **PRIVATE:** define que o atributo é privado.

Sintaxe: PRIVATE;

d) **PUBLIC:** define que o atributo é público.

Sintaxe: PUBLIC;

e) **REQUIRED:** define que o atributo é requerido.

Sintaxe: REQUIRED;

f) **TYPE:** define o tipo de dado do atributo. Entre os tipos suportados pelo Caché estão:

- **%Boolean;**
- **%Date;**
- **%Float;**
- **%Satus;**
- **%Time;**
- **%String;**

- **%Name.**

Sintaxe: TYPE = **TYPEDefinition;**

g) **INITIAL:** define uma expressão em COS ou um valor literal que é assumido como valor do atributo.

Sintaxe: INITIAL = **Expression;**

A4.1 method

Sintaxe: METHOD **MethodName (Pars) {MedthodDefinition}** ou

METHOD **MethodName FormalSpec (Pars) {MedthodDefinition}**

Define os métodos da classe.

MethodName define o nome do método.

FormalSpec define o separador padrão para os parâmetros formais desse método. O separador default é a “,” (vírgula).

Pars define os parâmetros formais do método. Esses parâmetros são passados ao método quando de sua chamada. **Pars** consiste de uma cadeia de caracteres no formato: **FormaParameter : FormalType = DefaultValue**, separadas pelo **FormalSpec.]**

MethodDefinition concite de um série de palavras reservadas para definição das características do método e também possui um sintaxe similar as de definições de classe. Dentre suas palavras reservadas estão:

a) **CALL:** define o nome do progrma que deve ser executado quando da chamada deste método.

Sintaxe: CALL = **ProgramName;**

b) **CLASSMETHOD:** define que o método é um método de classe.

Sintaxe: CLASSMETHOD;

c) **CODE:** Define o código COS do método.

Sintaxe: CODE = {**Bloco de Comandos**}

d) **FINAL:** define que o método é final e não pode ser sobreposto na subclasses.

Sintaxe: FINAL;

e) **PRIVATE:** define que o método é privado.

Sintaxe: PRIVATE;

f) **PUBLIC:** define que o método é publico.

Sintaxe: PUBLIC;

g) **RETURNTYPE:** define o tipo de dado do valor retornado pelo método. Quando RETURNTYPE não está presente na declaração do método ou recebe o valor nulo (“”) o método não retorna nenhum valor.

Sintaxe: RETURNTYPE = **DataType;**

h) **DESCRIPTION:** define uma descrição para o método.

Sintaxe: DESCRIPTION = “**Descrição do Método**”;

ANEXO 2 - DEFINIÇÃO DAS CLASSES

A2.1 ATRIBUTO

```
//*********************************************************************
//*        PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL          *
//*             ATRAVÉS DO REPOSITÓRIO DA  FERRAMENTA                  *
//*                     CASE SYSTEM ARCHITECT                          *
//* Autor: Danilo Kramel                                               *
//* Objetivo: Contém informações sobre os atributos de cada            *
//*           classe modelada no System Architect. Essas informações   *
//*           são adquiridas através do repositório do System Architect *
//*********************************************************************
unit C_Atributo;
interface
uses Classes, Sysutils;
Type
        TAtributo = class
        Private
                Name      : String;
                Acesso    : String;
                Tipo      : String;
                Tamanho   : integer;
                Derivado  : Boolean;
                Key       : Boolean;
        Public
                constructor CriaAtributo (PName, PAcesso, PTipo, PTamanho,
PDerivado, PKey : String);
                Destructor  LiberaAtributo;
                Procedure BuscaAtributo (Var PName, PAcesso, PTipo: String; Var
PTamanho : Integer; Var PDerivado, PKey : Boolean);
        end;
```

A2.2 BASE

```
//*********************************************************************
//*        PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL          *
//*             ATRAVÉS DO REPOSITÓRIO DA  FERRAMENTA                  *
//*                     CASE SYSTEM ARCHITECT                          *
//* Autor: Danilo Kramel                                               *
//* Objetivo: Criar os componetes de acesso ao repositório da          *
//*           ferramenta SA e ler a informações                        *
//*********************************************************************
unit C_Base;
interface
uses classes, dbtables, stdctrls, C_Classe, Messages, Sysutils, Forms, controls,
C_Relacao, C_Cdl;
Type
  TBase = class
  Private
      Classes : TList;
      Relacao : TList;
      Procedure CriaBase;
      Procedure LiberaBase;
      Function BuscaDescricao (PChave : String): string;
      Procedure LimpaListaClasse;

  Public
```

```
      constructor Cria;
      destructor Finaliza;
      Procedure GetDados;
      Function GetClasse: TStringList;
      Function GetRelacao : TStringList;
      Function GetListaClasse : TList;
      Function GetListaRelacao : TList;

  end;
implementation
var
    dbBase : TDatabase;
    qryPesquisa : TQuery;
    qryClasse: TQuery;
    Form : TForm;
    Descricao : TMemo;

{*******************************************************
*   Busca  a Descrição das Classes, Atributos e Métodos *
*******************************************************}
Function TBase.BuscaDescricao (PChave : String): string;
 Var
   Cont : integer;
begin
  Result := '';
  for cont := 0 to descricao.Lines.Count do
    if Descricao.Lines.Strings[cont] = PChave then
       Result := Descricao.Lines.Strings[cont+1]

end;

{*******************************************************
*   Cria os Componentes para Acessar a Base de dados    *
*******************************************************}
Procedure TBase.CriaBase;
begin
  // *** Base de Dados ***
  dbbase := nil;
  dbBase := TDatabase.Create(dbBase);
  dbBase.AliasName := 'TCC_SA';
  dbBase.DatabaseName := 'dbBase';
  dbBase.SessionName := 'Default';

  //*** Query Classe ***
  qryClasse := nil;
  qryClasse := TQuery.Create (qryClasse);
  qryClasse.DatabaseName := 'dbBase';

  //*** Query Pesquisa ***
  qryPesquisa := nil;
  qryPesquisa := TQuery.Create (qryPesquisa);
  qryPesquisa.DatabaseName := 'dbBase';
  qryPesquisa.SQL.Clear;

  Form := TForm.Create (Form);
  // *** Cria campo Memo para a descrição ***
  Descricao := TMemo.Create(Descricao);
  Descricao.Parent := Form;
end;

{*******************************************************
*     Libera o componetes de acesso ao repositório     *
*******************************************************}
Procedure TBase.LiberaBase;
```

```
begin
  dbBase.Free;
  qryClasse.Free;
  qryPesquisa.Free;
  Form.Free;
end;

{*******************************************************
*              Libera as classes  criadas              *
*******************************************************}
procedure TBase.LimpaListaClasse;
var
  i: Integer;
begin
  //*** Classe ***
  for i := 0 to Classes.Count - 1 do
     TClasse(Classes[i]).LiberaClasse;

  //*** Relação ***
  for i := 0 to Relacao.Count - 1 do
      TRelacao(Relacao[i]).LiberaRelacao;
end;

{*******************************************************
*                  Cria a classe Base                  *
*******************************************************}
constructor TBase.Cria;
begin
  inherited Create;
  Classes := TList.Create;
  Relacao := TList.Create;
  CriaBase;
end;

{*******************************************************
*                  Lipera a classe Base                *
*******************************************************}
destructor TBase.Finaliza;
begin
  LimpaListaClasse;
  LiperaBase;
  inherited Destroy;
end;

{*******************************************************
*                  Le o Repositório                    *
*******************************************************}
Procedure TBase.GetDados;
Var
  StrSql : String;
begin

  LimpaListaClasse;
  Classes.Clear;
  Relacao.Clear;

  dbBase.Open;
  //*** Classe ***
  qryclasse.SQL.clear;
  qryclasse.SQL.Add ('Select id, name, descriptn from Entity where class = "    3"
and Type = "   26"');
  qryClasse.Close;
  qryClasse.Open;
  with qryClasse do
```

```
  begin
    first;
    while not Eof do
    begin
       Descricao.Lines.Clear;
       Descricao.Lines.Add (FieldByname('descriptn').AsString);
       Classes.Add(TClasse.CriaClasse(FieldByname('Name').AsString,   BuscaDescricao
('[[[Abstract]]]'),BuscaDescricao                  ('[[[Derived]]]'),BuscaDescricao
('[[[Persistence]]]')));

       //*** Busca os Atributos e Metodos da Classe ***
       StrSql := 'Select en.id, en.name, en.type, en.descriptn ' +
                 'from entity en, relatn re  ' +
                 'where  en.id  =  re.id2  and  re.relation  =  14  and  re.id  =  '  +
intToStr(FieldByname('ID').AsInteger);

      qryPesquisa.SQL.Clear;
      qryPesquisa.SQL.Add(StrSql);
      qryPesquisa.open;
      qryPesquisa.First;
      while not qryPesquisa.Eof do
      begin
        Descricao.Lines.Clear;
        Descricao.Lines.Add (qryPesquisa.FieldByname('descriptn').AsString);
        case StrToInt(qryPesquisa.FieldByname('Type').AsString) of

        //Atributo
        133:       TClasse(Classes[Classes.Count       -       1]).InformaAtributo
(qryPesquisa.FieldByname('Name').AsString,        BuscaDescricao        ('[[[Delphi
Visibility]]]'),BuscaDescricao ('[[[Delphi Set Type]]]'),BuscaDescricao ('[[[Delphi
String Length]]]'),BuscaDescricao ('[[[Derived]]]'),BuscaDescricao ('[[[Key]]]'));

        //Metodo
        60:      TClasse(Classes[Classes.Count      -      1      ]).InformaMetodo
(qryPesquisa.FieldByname('Name').AsString,        BuscaDescricao        ('[[[DELPHI
Visibility]]]'),BuscaDescricao    ('[[[DELPHI    Return    Type]]]'),BuscaDescricao
('[[[DELPHI Method Type]]]'), BuscaDescricao ('[[[Formal Parameters]]]'));
        end;
        qryPesquisa.Next;
      end;
      qryPesquisa.Close;
      next;
    end;
  end;
  qryClasse.Close;

  //*** Tipos de Associações ***
  //*** Herança ***
  StrSql := 'Select ID, descriptn from entity where class = "    2" and Type = "
431"';
  qryPesquisa.SQL.Clear;
  qryPesquisa.SQL.Add(StrSql);
  qryPesquisa.open;
  qryPesquisa.First;
  while not qryPesquisa.Eof do
  begin
    Descricao.Lines.Clear;
    Descricao.Lines.Add (qryPesquisa.FieldByname('descriptn').AsString);
    Relacao.Add        (TRelacao.CriaRelacao        (BuscaDescricao       ('[[[Base
Class]]]'),BuscaDescricao ('[[[Derived Class]]]'), 1));
    qryPesquisa.Next;
  end;
  qryPesquisa.Close;
```

```
  //*** Associações ***
  StrSql := 'Select ID, ToAssc, FromAssc, descriptn from entity where class = "
2" and Type = "  376"';
  qryPesquisa.SQL.Clear;
  qryPesquisa.SQL.Add(StrSql);
  qryPesquisa.open;
  qryPesquisa.First;
  while not qryPesquisa.Eof do
  begin
    Descricao.Lines.Clear;
    Descricao.Lines.Add (qryPesquisa.FieldByname('descriptn').AsString);
    if       (qryPesquisa.FieldByname('ToAssc').AsInteger       =       8)       and
(qryPesquisa.FieldByname('FromAssc').AsInteger = 9) then
    begin
      //Agregaçao
      Relacao.Add        (TRelacao.CriaRelacao        (BuscaDescricao        ('[[[To
Class]]]'),BuscaDescricao ('[[[From Class]]]'), 3));
    end else begin
      // Associação
      Relacao.Add        (TRelacao.CriaRelacao        (BuscaDescricao        ('[[[To
Class]]]'),BuscaDescricao ('[[[From Class]]]'), 2));
    end ;
    qryPesquisa.Next;
  end;
  qryPesquisa.Close;
  dbbase.Close;
end;

{*******************************************************
*            Busca as Informações da Classe            *
*******************************************************}
Function TBase.GetClasse : TStringList;
var
  cont        : integer;
  AuxLeClasse : TStringList;
begin
  AuxLeClasse:= TStringList.Create;
  AuxLeClasse.Clear;
  For cont:= 0 to classes.Count -1 do
    AuxLeClasse.Add (TClasse(Classes[Cont]).LeClasse);

  Result := AuxLeClasse;
end;

{*******************************************************
*       Busca as Informações dos Relacionamentos       *
*******************************************************}
Function TBase.GetRelacao : TStringList;
var
  cont        : integer;
  AuxLeRelacoes : TStringList;
begin
  AuxLeRelacoes:= TStringList.Create;
  AuxLeRelacoes.Clear;
  For cont:= 0 to Relacao.Count -1 do
    AuxLeRelacoes.Add (TRelacao(Relacao[Cont]).LeRelacao);

  Result := AuxLeRelacoes;

end;

{*******************************************************
*          Busca os objetos da classe  Classe          *
*******************************************************}
```

```
Function TBase.GetListaClasse : TList;
begin
  Result := Classes;

end;


{*****************************************************
*         Busca os objetos da classe Relacao            *
*****************************************************}
Function TBase.GetListaRelacao : TList;
begin
  Result:= Relacao;

end;
end.
```

A2.3 CLASSE

```
//*************************************************************
//*        PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL   *
//*             ATRAVÉS DO REPOSITÓRIO DA  FERRAMENTA           *
//*                      CASE SYSTEM ARCHITECT                  *
//* Autor: Danilo Kramel                                        *
//* Objetivo: Contém as informação das CLASSES, obtidas         *
//*           através dos repositório do System Architect (SA)  *
//*************************************************************
unit C_Classe;
interface
uses classes, C_Atributo, C_Metodo, Messages, Sysutils;
const
        TamVetor = 100;
type
        TClasse = class
        Private
                Name        : String ;
                Abstrato    : Boolean;
                Derivado    : Boolean;
                Persistente : String ;
                IntAtrib    : integer;
                IntMetod    : integer;
                IntAux      : integer;
                Atributo    : array[1..TamVetor] of TAtributo;
                Metodo      : array[1..TamVetor] of TMetodo;

        Public
                constructor CriaClasse(PName,PAbstrato,PDerivado, PPersistente :
String);
                destructor LiberaClasse;
                Procedure BuscaClasse (Var PName, PPersistente : String;  Var
PAbstrato, PDerivado : Boolean);
                Function  LeCLasse : String;
                Procedure InformaAtributo (PName, PAcesso, PTipo, PTamanho,
PDerivado, PKey : String);
                Procedure InformaMetodo (PName, PAcesso, PRetorno, PTipo,
PParametro : String);
                Procedure Start;
                Procedure Next;
                Function  EofAtributo: Boolean;
                Function  EofMetodo : Boolean;
                Procedure GetAtributo (Var PName, PAcesso, PTipo: String; Var
PTamanho : Integer; Var PDerivado, PKey : Boolean);
```

```
               Procedure GetMetodo (Var PName, PAcesso, PRetorno, PTipo,
PParametro : String);
        end;
```

A2.4 CDL

```
//*********************************************************************
//*        PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL          *
//*             ATRAVÉS DO REPOSITÓRIO DA  FERRAMENTA                  *
//*                      CASE SYSTEM ARCHITECT                         *
//* Autor: Danilo Kramel                                               *
//* Objetivo: gerar o código em CDL                                    *
//*********************************************************************
unit C_Cdl;

interface
uses classes, C_Classe, C_Relacao, Sysutils;
Type
  TCdl = class
  Private
      Classes : TList;
      Relacao : TList;
      Function IsFinal(PName: String): Boolean;
      Function GetSuper(PName: String): String;
      Function GetAgregacao(PName: String): String;
      Function GetAssociacao(PName: String): String;
  Public
      constructor CriaCDL (PClasses, PRelacao : TList);
      Destructor Finaliza;
      Function GeraCodigo (PCont : Integer): string;
  end;

implementation

{*********************************************************
 *                     Cria a Classe CDL                 *
 *********************************************************}
constructor TCdl.CriaCDL (PClasses, PRelacao : TList);
begin
  inherited Create;
  Classes := PClasses;
  Relacao := PRelacao;
end;

{*********************************************************
 *                     Libera a Classe CDL               *
 *********************************************************}
Destructor TCdl.Finaliza;
begin
  inherited destroy;
end;

{*************************************************************
 *      Retorna verdadeiro caso não exista classe filha      *
*************************************************************}
Function TCdl.IsFinal(PName: String): Boolean;
var
 cont: integer;
begin
   IsFinal := True;
   for cont := 0 to Relacao.Count -1 do
      //Retorna o nome da classe Pai
```

```
      If TRelacao(Relacao[cont]).BuscaRelacao (1,'P') = PName then
      begin
        IsFinal := False;
        exit;
      end;
end;

{*********************************************************
 *          Retorna o nome da classe pai caso exista       *
//*********************************************************}
Function TCdl.GetSuper(PName: String): String;
var
 cont: integer;
begin
   GetSuper := '%RegisteredObject';
   for cont := 0 to Relacao.Count -1 do
      //Retorna o nome da classe filha
      If TRelacao(Relacao[cont]).BuscaRelacao (1,'F') = PName then
      begin
        //Retorna o nome da classe Pai
        GetSuper := TRelacao(Relacao[cont]).BuscaRelacao (1,'P');
        exit;
      end;
end;

{*****************************************************
 *     Retorna todos as agregações da classe PName     *
 *****************************************************}
Function TCdl.GetAgregacao(PName: String): String;
var
 cont      : integer;
 AuxClasse : String;
 Atributos : String;
begin
   GetAgregacao := '';
   Atributos := '';

   for cont := 0 to Relacao.Count -1 do
      //Retorna o nome da classe Pai
      If TRelacao(Relacao[cont]).BuscaRelacao (3,'P') = PName then
      begin
        //Retorna o nome da classe Filha
        AuxClasse := TRelacao(Relacao[cont]).BuscaRelacao (3,'F');
        Atributos := Atributos + '   list attribute Agre'+ AuxClasse + ' {type = '+
AuxClasse +'; }' + chr(13);
      end;

   GetAgregacao := Atributos;
end;

{*****************************************************
 *    Retorna todas as Associações da classe PName    *
 *****************************************************}
Function TCdl.GetAssociacao(PName: String): String;
var
 cont      : integer;
 AuxClasse : String;
 Atributos : String;
begin
   GetAssociacao := '';
   Atributos := '';

   for cont := 0 to Relacao.Count -1 do
      //Retorna o nome da classe Pai
```

```
      If TRelacao(Relacao[cont]).BuscaRelacao (2,'P') = PName then
      begin
        //Retorna o nome da classe Filha
        AuxClasse := TRelacao(Relacao[cont]).BuscaRelacao (2,'F');
        Atributos := Atributos + '   attribute Ass'+ AuxClasse + ' {type = '+
AuxClasse +'; }' + chr(13);
      end;

   GetAssociacao := Atributos;
end;

{*************************************************
 *               Gera o Código em CDL            *
 *************************************************}
Function TCdl.GeraCodigo(PCont : integer): string;
Var
   wClasse : TClasse;

     //*** Classe ***
   Name        : String ;
   Abstrato    : Boolean;
   Derivado    : Boolean;
   Persistente : String ;
   StrClasse   : String;

   // *** Atributo ***
   AName        : String ;
   AAcesso      : String ;
   ATipo        : String ;
   ATamanho     : integer;
   ADerivado    : Boolean;
   AKey         : Boolean;
   StrAtributo  : String;
   StrIndice    : String;

   //*** Método ***
   MName,
   MAcesso,
   MRetorno,
   MTipo,
   MParametro,
   StrMetodo : String;
begin

  result := '';
  wClasse := (TClasse(classes[PCont]) as TClasse);
  wclasse.BuscaClasse(Name, Persistente, Abstrato, Derivado);
  StrClasse := '';
  StrClasse := 'Drop Class '+ Name + ';' + chr(13) ;
  StrClasse := StrClasse + 'Create Class ' + Name + chr(13) + '{' + chr(13);
  if IsFinal (Name) then
    StrClasse := StrClasse + '   final;' + chr(13);

  StrClasse := StrClasse + '   super = ' + GetSuper(Name) + ' ;' + chr(13);
  if Persistente <> '' then
    StrClasse := StrClasse + '   '+ Persistente + '; ' + chr(13);

  StrAtributo := '';
  StrIndice := '';
  StrAtributo := GetAgregacao(Name) + GetAssociacao(Name);

  //*** Atributos ***
  wclasse.Start;
  while not wclasse.EofAtributo do
```

```
  begin
    AName := '';
    AAcesso:= '';
    ATipo:= '';
    ATamanho:= 0;
    ADerivado:= False;
    AKey:= False;
    wClasse.GetAtributo (AName, AAcesso, ATipo,ATamanho, ADerivado, AKey);
    If AKey then
      StrIndice := StrIndice + '    Index ' + AName + 'Index { Attributes = ' +
AName + '; Unique; }' + chr(13);

    If (ATipo = 'String') and  (ATamanho <> 0) then
      StrAtributo := StrAtributo + '   attribute ' + AName + ' {type = %'+ATipo+'
(MAXLEN = '+ IntToStr(ATamanho) + '); '
    else
      StrAtributo := StrAtributo + '   attribute ' + AName + ' {type = %'+ATipo+';
';

    If AAcesso <> '' then
      StrAtributo := StrAtributo + AAcesso + '; ';

    StrAtributo := StrAtributo + '   }' +chr(13);
    wclasse.Next;
  end;

  //*** Métodos ***
  StrMetodo := '';
  wClasse.Start;
  while not wClasse.EofMetodo do
  begin
    MName := '';
    MAcesso := '';
    MRetorno := '';
    MTipo := '';
    MParametro := '';
    wClasse.GetMetodo (MName, MAcesso, MRetorno, MTipo, MParametro);
    StrMetodo := StrMetodo + '   Method ' + MName + ' ('+ MParametro + '){' +
chr(13);
    IF MTipo = 'Function' then
      StrMetodo := StrMetodo + '      returntype = %' + MRetorno + '; ' + chr(13);

    StrMetodo := StrMetodo + '      classmethod = 0; ' + chr(13);
    StrMetodo := StrMetodo + '      final; ' + chr(13);
    if MAcesso <> '' then
      StrMetodo := StrMetodo + '       ' + MAcesso + ';' + chr(13);

    StrMetodo := StrMetodo + '      sqlproc = 0; '+ chr(13);
    StrMetodo := StrMetodo + '      code = {  }' + chr(13);
    StrMetodo := StrMetodo + '   }' + chr(13);
    wClasse.Next;
  end;
  Result := StrClasse + chr(13) + StrIndice + chr(13) + StrAtributo + chr(13) +
StrMetodo + chr(13) + ' }' + Chr(13) + Chr(13);
end;
end.
```

A2.5 METODO

```
//*********************************************************************
//*        PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL          *
//*             ATRAVÉS DO REPOSITÓRIO DA  FERRAMENTA                  *
//*                     CASE SYSTEM ARCHITECT                          *
//* Autor: Danilo Kramel                                               *
//* Objetivo: Contém informações sobre os métodos de cada classe       *
//*           modelada no System Architect. Essas informações são      *
//*           adquiridas através do repositório do System Architect    *
//*********************************************************************
unit C_Metodo;
interface
uses Classes;
Type
        TMetodo = class
        Private
                Name        : String;
                Acesso      : String;
                Retorno     : String;
                Tipo        : String;
                Parametro   : String;
        Public
                constructor CriaMetodo (PName, PAcesso, PRetorno, PTipo, PParametro
: String);
                Destructor LiberaMetodo;
                Procedure BuscaMetodo (Var PName, PAcesso, PRetorno, PTipo,
PParametro : String);
end;
```

A2.6 RELACAO

```
//*********************************************************************
//*        PROTÓTIPO DE SOFTWARE PARA A GERAÇÃO DE CÓDIGO CDL          *
//*             ATRAVÉS DO REPOSITÓRIO DA  FERRAMENTA                  *
//*                     CASE SYSTEM ARCHITECT                          *
//* Autor: Danilo Kramel                                               *
//* Objetivo: Utilizado para determinar o relacionamento entre as      *
//*           classes modeladas no System Architect                    *
//*********************************************************************
unit C_Relacao;
interface
uses Classes;
Type
        TRelacao = class
        Private
                ToClass   : String ;
                FromClass : String ;
                Tipo      : Integer;
        Public
                constructor CriaRelacao (PToClass, PFromClass :  string ; PTipo :
Integer);
                destructor LiberaRelacao;
                Function BuscaRelacao (PTipo : Integer; PClasse: String) : String;
                Function LeRelacao : String;

        end;
```

ANEXO 3 - CÓDIGO FONTE EM CDL

```
Drop Class Curso;
Create Class Curso
{
   final;
   super = %RegisteredObject ;

    Index cdCursoIndex { Attributes = cdCurso; Unique; }

   attribute cdCurso {type = %Integer; Private;    }
   attribute nmCurso {type = %String (MAXLEN = 30); Private;    }
   attribute tpCurso {type = %String (MAXLEN = 30); Private;    }

   Method VerificaCurso (){
      returntype = %String;
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }
   Method CriaCurso (PcdCurso %integer, PnmCurso ){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }
   Method LiberaCurso (){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }

 }

Drop Class Matricula;
Create Class Matricula
{
   final;
   super = %RegisteredObject ;

    Index cdMatriculaIndex { Attributes = cdMatricula; Unique; }

   list attribute AgreDiscMatri {type = DiscMatri; }
   attribute cdMatricula {type = %Integer; Private;    }
   attribute dsSemestre {type = %String (MAXLEN = 15); Private;    }
   attribute dtMatricula {type = %String (MAXLEN = 10); Private;    }

   Method LiberaMatricula (){
      classmethod = 0;
```

```
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }
   Method CriaMatricula (PcdMatriculas %integer, PdsSemestre ){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }

 }

Drop Class Disciplina;
Create Class Disciplina
{
   final;
   super = %RegisteredObject ;

    Index cdDisciplinaIndex { Attributes = cdDisciplina; Unique; }

   attribute AssCurso {type = Curso; }
   attribute cdDisciplina {type = %Integer; Private;    }
   attribute dsDisciplina {type = %String (MAXLEN = 30); Private;    }

   Method CriarDisciplina (PcdDisciplina %integer, PdsDisciplina ){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }
   Method LiberaDisciplina (){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }

 }

Drop Class DiscMatri;
Create Class DiscMatri
{
   final;
   super = %RegisteredObject ;


   attribute AssDisciplina {type = Disciplina; }

   Method CriaDisciplina (){
      classmethod = 0;
      final;
       Public;
```

```
      sqlproc = 0;
      code = {  }
   }
   Method LiberaDisciplina (){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }

 }

Drop Class Aluno;
Create Class Aluno
{
   super = %RegisteredObject ;

    Index cdAlunoIndex { Attributes = cdAluno; Unique; }

   attribute AssCurso {type = Curso; }
   attribute AssMatricula {type = Matricula; }
   attribute cdAluno {type = %Integer; Private;    }
   attribute nmAluno {type = %String (MAXLEN = 30); Private;    }
   attribute dtNascto {type = %String (MAXLEN = 10); Private;    }

   Method CriaAluno (PcdAluno %integer, PnmAluno ){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }
   Method LiberaAluno (){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }

 }

Drop Class Graduacao;
Create Class Graduacao
{
   final;
   super = Aluno ;


   attribute dtVestibu {type = %String (MAXLEN = 10); Private;    }

   Method Liberagraduacao (){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
```

```
      code = {  }
   }
   Method CriaGraduacao (PdtVestibul %string){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }

 }

Drop Class PosGraduacao;
Create Class PosGraduacao
{
   final;
   super = Aluno ;


   attribute dsCurso {type = %String (MAXLEN = 30); Private;    }
   attribute dtAnoFormacao {type = %String (MAXLEN = 4); Private;    }

   Method LiberaPosgraduacao (){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }
   Method CriaPosgraduacao (PdtAno %string, PdsCurso %string){
      classmethod = 0;
      final;
       Public;
      sqlproc = 0;
      code = {  }
   }

 }
```

REFERÊNCIAS BIBLIOGRÁFICAS

[AHO1995] AHO, Alfred V. **Compiladores princípios, técnicas e ferramentas.** Rio de Janeiro : Livros Técnicos e Cientificos Editora S.A, 1995.

[BAE1999] BAEHR Junior, Ivo. **Protótipo de sistema para gerenciamento de ordens de serviço acessando um banco de dados orientado a objetos e um banco de dados relacional**. Blumenau, 1999. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

[CAN2000] CANTÚ, Marco. **Dominando o Delphi 5 – A Bíblia**. Trad. de João E.N Tortello. São Paulo : Markon Books, 2000.

[CHO1999] CHOOSE, Technologies. **Roteiro de avaliação da ferramenta CASE System Architect/2001**. Choose, 1999.

[DAT1991] DATE, C.J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro : Campus, 1991.

[FIS1990] FISHER, Alan S. **Case: utilização de ferramentas para desenvolvimento de software**. Trad. de Info-Rio. Rio de Janeiro : Campus, 1990.

[FUR1998] FURLAN, José Davi. **Modelagem de objetos através da UML - The unified modeling language**. São Paulo : Makron Books, 1998.

[GAN1990] GANE, Chris. **CASE**. Trad. de Mauro Lando. Rio de Janeiro : LTC Livros Técnicos e Científicos, 1990.

[GON2000] GONÇALVES, Glaura Inácio; PESENTE, Graziela Rolim; LIMA, Rafael. **Banco de dados orienta a objetos** 1996**.** Endereço Eletrônico : http://www.dsc.ufpb.br/~helder/java/cap_2.html. Data de consulta : 01/08/2000.

[HAC1993] HACKATHORN, Richard D. **Conectividade de banco de dados empresariais**. Trad. de Flávio Eduardo Frony Morgado. Rio de Janeiro : Infobook, 1993.

[HEU2000] HEUSER, Carlos Alberto. **Projeto de banco de dados**. Porto Alegre : Sagra Luzzatto, 2000.

[INT1999] INTERSYSTEMS CORPORATION. **Caché development guide**. InterSystems, 1999.

[IPS1999] IPSUM. **Caché – the e-dbms** 1999**.** Endereço Eletrônico : http://www.ipsum.com.br/cache01.htm. Data Consulta : 19/09/2000.

[JOS1987] JOSÉ Neto, João. **Introdução à compiladores**. Rio de Janeiro : Livros Técnicos e Cientificos Editora S.A, 1987.

[KHO1994] KHOSHAFIAN, Setrang. **Banco de dados orientado a objetos**. Trad. de Trype Informática. Rio de Janeiro : Infobook, 1994.

[KOR1991] KORTH, Henry F; SILBERSCHATZ, Abraham. **Database system concepts**. New York : Ed. McGraw-Hill, 1991.

[LEI1980] LEITE, Leonardo Lellis Pereira. **Introdução aos sistemas de gerência de banco de dados**. São Paulo : Ed. Edgard Blücher, 1980.

[MAR1995] MARTIN, James. **Análise e projetos orientados a objeto**. Trad. de José Carlos Barbosa dos Santos. São Paulo : Makron Books, 1995.

[NAS1999] NASSU, Eugênio A.; SETZER, Valdemar W. **Banco de dados orientado a objetos.** São Paulo : Edgard Blücher, 1999.

[POP1998] POPKIN SOFTWARE SYSTEMS INC. **System architec user guide**. New York : Popkin, 1998.

[POP1999] POPKIN Software Inc. **Tutorial System Architect/2001**. Popkin, 1999.

[PRE1995] PRESSMAN, Roger S. **Engenharia de software**. Trad. de José Carlos Barbosa dos Santos. São Paulo : Makron Book, 1995.

[RAF2000] RAFARTS INFORMÄTICA. **Case** 2000. Endereço Eletrônico : http://www.orbita.starmedia.com/~rafarts99/tutoriais.html. Data Consulta: 05/09/2000.

[RAM1997] RAMAKRISHNAN, Raghu. **Database management systems**. New York : Ed. McGraw-Hill, 1997.

[ROC1996] ROCHA, Helder L.S. **Programação orientada a objetos** 1996 Endereço Eletrônico : http://www.dsc.ufpb.br/~helder/java/cap_2.html. Data de consulta : 05/08/2000.

[SAL1992] SALGADO, Ana Carolina. **Sistema de hipermídia – hipertexto e banco de dados.** Porto Alegre : Instituto de Informática de UFRGS, 1992.

[SAL1999] SALDANHA, Emerson Batista. **Protótipo de uma ferramenta de apoio a migração de especificação estruturada para especificação por objetos.** Blumenau, 1999. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

[TAU1997] TAURION, Cesar. Developers Magazine. **Banco de dados universais: um novo impulso à tecnologia oo**. Rio de Janeiro, v. 2, n.11, p. 10-11, outubro,1997.

[URR1999] URRESTI, Hugo Ricardo. Developers Magazine. **Banco de dados: a renovação da plataforma de ti**. Rio de Janeiro, v. 4, n.38, p. 18-19, outubro,1999.

[WIN1993] WINBLAD, Annl; EDWARDS, Samuel D.; KING, David R. **Software orientado ao objeto.** Trad. de Denise de Souza Boccia. São Paulo : Makron Book, 1993.