UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO COSTA CAMPOS

UMA IMPLEMENTAÇÃO DO SERVIÇO DE PERSISTÊNCIA CORBA PARA INTEGRAÇÃO DE BASES DE DADOS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do Grau de Mestre em Ciência da Computação

JOÃO BOSCO MANGUEIRA SOBRAL

Florianópolis, Novembro 2002

UMA IMPLEMENTAÇÃO DO SERVIÇO DE PERSISTÊNCIA CORBA PARA INTEGRAÇÃO DE BASES DE DADOS

MARCELO COSTA CAMPOS

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Fernando Álvaro Ostuni Gauthier
(Coordenador do PPGCC)

Prof. João Bosco Mangueira Sobral, Dr.
(Orientador)

Banca Examinadora

Prof. João Bosco Mangueira Sobral, Dr.

Prof. Jorge Muniz Barreto, Dr.

Prof. Bernardo Gonçalves Riso, Dr.

"Temos por PRINCÍPIO evoluir o homem,
porque não acreditamos que o homem seja produto do meio,
mas sim, que o meio é produto do homem"

(Dr. CELSO CHARURI)

Para mim, o ser mais importante é o nosso Senhor Jesus Cristo, devemos todo o nosso amor e agradecimento a Ele, que sempre esteve e estará ao nosso lado ajudando e abençoando todos os momentos de nossas vidas.

O meu agradecimento especial vai para toda a minha família, em especial aos meus pais, irmãos, avós, tios, primos e para Priscilla a minha futura esposa. Não deixo de fazer referências aos funcionários da Faculdade Cândido Rondon, professores do mestrado da Universidade Federal de Santa Catarina, aos meus colegas de sala e para o professor Cristiano Maciel, João Bosco Sobral, meu orientador, e Daniela Claro.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Aplication Programmer Interface*
ACID	(Atomicidade, Concorrência, Isolamento e Durabilidade)
BDR	*Banco de Dados Relacional*
BDOR	*Banco de Dados Objeto Relacional*
BDOO	*Banco de Dados Orientado a Objetos*
COSS	*Common Object Services Specifications*
COS	*Common Object Services*
CORBA	*Common Object Request Broker Architecture*
GOP	*Gateway-Based Object Persistence*
IDL	*Interface Definition Language*
JDBC	*Java DataBase Connectivity*
ODBC	*Open DataBase Connectivity*
OMG	*Object Management Group*
ORB	*Object Request Broker*
ORDBMS	*Object-Relational DBMS*
OODBMS	*Object-Oriented DBMS*
POS	*Persistent Object Service*
PSDL	*Persistent State Definition Language*
PSS	*Persistent State Service*
SGBD	Sistema Gerenciador de Banco de Dados
SQL	*Structured Query Language*

RESUMO

Este trabalho esta inserido no contexto de Objetos Distribuídos e versa sobre o desenvolvimento de um serviço de persistência de objetos, apropriado ao contexto da arquitetura CORBA. A implementação do serviço de persistência utiliza produtos que estão em conformidade com o padrão CORBA e foi desenvolvido baseando-se na linguagem de programação JAVA.

O modelo do Serviço de Persistência implementado apresenta-se como o mediador entre a aplicação e o sistema de armazenamento de dados. O *Persistent State Service* (PSS) especificado pelo padrão CORBA, procura facilitar o gerenciamento de mudanças (alteração da estrutura de um objeto, novos serviços e novas regras de negócios) evitando a necessidade de re-compilação dos objetos de serviço. Assim é que, a possibilidade de interoperar estas distintas tecnologias utilizando as características de um sistema distribuído, torna este trabalho capaz de identificar as vantagens encontradas nesse tipo de sistema, tais como: maior escalabilidade, desempenho, interoperabilidade, segurança e rapidez em suas transações, seguindo o Serviço de Persistência de Objetos.

Abordar-se-a implementação do Serviço de Persistência de Objetos PSS, que é objeto de pesquisa deste trabalho, com ênfase na utilização da arquitetura CORBA e desenvolvido na linguagem de programação JAVA. Possibilitando a interoperabilidade da aplicação entre distintas arquiteturas.

ABSTRACT

This inserted work this in the Distributed Object context and turns on the development of a service of object persistence, appropriate to the context of architecture CORBA. The implementation of the persistence service uses products that are in compliance with standard CORBA and were developed being based on the programming language JAVA

The model of the implemented Service of Persistence is presented as the mediator between the application and the system of storage of data. The Persistent State Service (PSS) specified by standard CORBA, looks for to facilitate the management of changes (alteration of the structure of an object, new services and new business-oriented rules) preventing the recompilation necessity of service objects. Thus it is that, the possibility of interoperar these distinct technologies using the characteristics of a distributed system, becomes this work capable to identify the advantages found in this type of system, such as: escalabilidade, performance, interoperabilidade, security and rapidity in its transactions, following the Service of Object Persistence.

To approach it implementation of the Service of Object Persistence PSS, that is object of research of this work, with emphasis in the use of developed architecture CORBA and in the programming language JAVA. Making possible the interoperabilidade of the application between the distinct architectures and available database.

1. INTRODUÇÃO

O presente trabalho situa-se na área de computação com seus respectivos objetos distribuídos e versa sobre o desenvolvimento de um serviço de persistência de objetos, apropriado ao contexto da arquitetura CORBA.

O fato de que uma aplicação cria objetos em tempo de execução e que não é possível manter esses objetos criados na memória de um computador por um tempo ilimitado e, que após a finalização da execução de uma aplicação, pode causar a necessidade de se armazenar esses objetos em meios físicos e seguros como banco de dados. Isso faz com que esse armazenamento proporcione a sobrevivência dos mesmos após a execução da aplicação, caracterizando assim, o que vem a ser a persistência de objetos.

Um objeto é dito persistente quando o seu ciclo de vida vai além dos limites da aplicação que o criou. Isto significa que o objeto deve ser gravado em um meio de armazenamento como, por exemplo, um arquivo, um banco de dados relacional (BDR), um banco de dados objeto relacional (BDOR) ou num banco de dados orientado a objetos (BDOO).

Existem alguns serviços de persistência de objetos como: o JDBC (*Java DataBase Connectivity*) em Java e o PSS (*Persistent State Service*) na arquitetura CORBA. Neste trabalho será abordado o PSS, no sentido de se estudar o serviço de persistência CORBA. Ressalta-se que o PSS fornece uma interface única para armazenar o estado persistente dos objetos em uma variedade de dispositivos de armazenamento, tais como: BDR, BDOR, BDOO e arquivos, PEREIRA (2000).

A motivação deste trabalho está no estudo do Serviço de Persistência CORBA, voltado para a interoperabilidade nas diversas tecnologias de armazenamento.

Um objeto distribuído é essencialmente um artefato de *software*, que pode interoperar com outros objetos distribuídos através de sistemas operacionais, redes, linguagens, aplicações, ferramentas e equipamentos diversos. No sentido da interoperabilidade

existe a tendência de se construir sistemas computacionais abertos utilizando objetos distribuídos, ORFALI (1997).

Os objetos distribuídos compõem aplicações distribuídas. Esses objetos são desenvolvidos de acordo com um modelo, com as características de herança. Daí o fato de os mesmos possuírem uma característica fundamental para desenvolvedores de aplicativos, a reutilização. Pois é possível reutilizar objetos de um outro sistema desenvolvido em outra linguagem de programação. Para serem reutilizados, esses objetos devem ser persistidos, ou seja, armazenados.

1.1. Trabalhos Correlatos

Alguns trabalhos já publicados, relacionados à persistência de objetos, apresentam o seguinte contexto histórico:

Em LINS (2000), encontra-se o desenvolvimento de Sistemas Persistentes em Java com JDBC, onde são colocadas as vantagens e desvantagens na utilização da interface JDBC para desenvolver sistemas com um serviço de persistência.

Já em COSTA (2000), vê-se o desenvolvimento de aplicações distribuídas com Java e CORBA é abordado visando a integração com bancos de dados relacional e orientados a objetos.

Em PEREIRA (2000), observa-se uma nova abordagem para persistência de objetos em ambiente distribuído, tendo como base o serviço de persistência PSS especificado pela OMG. O serviço desenvolvido tem o propósito de integrar tecnologias de bancos de dados, além de reduzir o tempo de desenvolvimento e a complexidade das aplicações.

1.2. Organização do Trabalho

O presente trabalho mostra como uma aplicação com objetos JAVA, pode usar uma implementação do PSS, para a sua utilização em diversos sistemas gerenciadores de banco de dados, independente de fornecedores (MySQL, SQL Server, Oracle 9i, DB2, CACHÉ, entre outros). Perante a diversidade de bancos de dados existentes, foram escolhidos para a realização deste estudo, o banco de dados relacional SQL Server 2000 e o banco de dados objeto relacional CACHÉ 4.

O presente trabalho está organizado em seis capítulos. No segundo capítulo faz-se a apresentação dos fundamentos técnicos sobre persistência de objetos. O terceiro capítulo apresenta os fundamentos da arquitetura CORBA, ressaltando o que é necessário para o desenvolvimento deste trabalho. No quarto capítulo é enfocado o estudo sobre o Serviço de Persistência CORBA, proporcionado pelo PSS. O quinto capítulo contém uma implementação do serviço de persistência de objeto PSS, utilizado numa aplicação CORBA. Finalmente, no sexto capítulo são apresentadas as conclusões deste trabalho.

2. PERSISTÊNCIA DE OBJETOS

Este capítulo descreve a conceituação básica sobre persistência de objetos. Nele são abordados os conceitos básicos do serviço de persistência, finalidade da persistência, arquitetura do serviço e utilização de persistência para aplicações JAVA.

2.1. Conceitos Básicos

O serviço de persistência de objetos define uma interface única para o armazenamento persistente do estado dos objetos nos diversos meios de armazenamento, incluindo Bancos de Dados Orientado a Objeto (ODBMS), Bancos de Dados Relacionais, Bancos de Dados Objeto-Relacional e também, arquivos simples.

Este serviço de persistência é uma especificação padronizada para que as aplicações possam armazenar e restaurar os objetos nestes meios de armazenamento. Este padrão de especificação vem impactar com as tradicionais aplicações de fornecedores para o armazenamento de objetos em banco de dados. Entretanto, os fornecedores das aplicações para armazenamento em banco de dados orientados a objetos visualizaram a real necessidade de agregar esta especificação em suas interfaces e processos, SESSIONS (1996).

2.2. Estado de um Objeto

O estado do objeto pode ser considerado de duas partes: o estado dinâmico, que está tipicamente na memória e não fazendo parte do ciclo de vida do objeto como um todo e o estado persistente, usado para construir o estado dinâmico, MCCARTY (1999).

2.3. Meios de Armazenamento de Objetos

2.3.1. Arquivos Flat

Um arquivo é uma estrutura de dados residente em memória auxiliar, que consiste num conjunto de informação estruturada e em unidades de acesso denominadas registros,

todos do mesmo tipo e em número indeterminado, ALCALDE (1991). Um registro lógico, ou simplesmente registro é cada um dos componentes do arquivo, contendo o conjunto de informações que são tratadas de forma unitária. É constituído por um ou mais elementos denominados campos, que podem ser de diferentes tipos e que por sua vez podem ser compostos por sub-campos.

Se um arquivo contém a informação de um conjunto de indivíduos ou objetos, os seus registros contêm a informação de cada um deles e os campos, os diferentes dados que a compõem, ALCALDE (1991).

Um registro físico ou bloco correspondem à quantidade de informação que se transfere em cada operação de acesso (leitura ou gravação).

Convém distinguir claramente os conceitos de registros lógicos e registros físicos. A diferença é que enquanto o tamanho e o formato do registro lógico são definidos pelo programador, o tamanho do registro físico vem dado pelas características físicas do computador utilizado.

Em geral, um registro físico (bloco) pode conter um ou mais registros lógicos, mas também pode acontecer de um registro lógico ocupar mais de um registro físico. No primeiro caso, diz-se que os registros estão em blocos, denominando-se fator de blocagem o número de registros lógicos que cada registro físico contém.

Para poder selecionar um registro do conjunto que compõe o arquivo, necessita-se de um dado identificador que o diferencie dos demais. Denomina-se *campo-chave* o campo especial do registro que serve para identificá-lo. Alguns arquivos não têm campo-chave nos seus registros, enquanto outros podem ter vários, recebendo estes-o nome de chave primária ou secundária, ALCALDE (1991).

As principais características dessa estrutura de dados são as seguintes:

- *Residência em suporte de informação externo*, também chamado de memória secundária ou auxiliar, como as fitas e os discos magnéticos.
- *Independência com relação ao programa*, o que significa que a vida do arquivo não está limitada pela vida do programa que o criou e também

que em diferentes momentos pode-se fazer uso do mesmo arquivo em diferentes programas.

- *Permanência das informações armazenadas*, ou seja, a informação contida num arquivo não desaparece quando o computador é desligado, diferentemente das informações armazenadas na memória principal.
- *Grande capacidade de armazenamento*, sendo esta capacidade teoricamente ilimitada; por outro lado, as estruturas de dados residentes na memória principal têm o seu tamanho limitado pela sua capacidade.

2.3.2. Banco de Dados

A tecnologia aplicada aos métodos de armazenamento de informações vem crescendo e gerando um impacto cada vez maior no uso de computadores, em qualquer área em que os mesmos podem ser aplicados, TEIXEIRA (1996).

Um banco de dados pode ser definido como um conjunto de dados devidamente relacionados. Por dados compreende-se como fatos conhecidos que podem ser armazenados e que possuem um significado implícito. Porém, o significado do termo banco de dados é mais restrito que simplesmente a definição dada acima. Um banco de dados possui as seguintes propriedades:

- um banco de dados é uma coleção lógica coerente de dados com um significado inerente;
- um banco de dados é projetado, construído e preenchido com dados para um propósito específico;
- um banco de dados possui um conjunto pré-definido de usuários e aplicações;
- um banco de dados representa algum aspecto do mundo real, o qual é chamado de mini-mundo, e qualquer alteração efetuada no mini-mundo é automaticamente refletida no banco de dados.

Um banco de dados pode ser criado e mantido por um conjunto de aplicações desenvolvidas especialmente para a tarefa de armazenagem de dados ou por um Sistema

Gerenciador de Banco de Dados (SGBD). Um SGBD permite aos usuários criarem e manipularem bancos de dados de propósito geral. O conjunto formado por um banco de dados, mais as aplicações que manipulam o mesmo é chamado de Sistema Gerenciador de Banco de Dados, TEIXEIRA (1996).

2.3.2.1. Banco de Dados Relacional

O modelo de dados relacional representa os dados através de um conjunto de relações (tabelas). Estas relações contêm informações sobre entidades e relacionamentos existentes no domínio da aplicação utilizada como alvo para a modelagem. Informalmente uma relação pode ser considerada como uma tabela de valores, onde cada linha desta tabela representa uma coleção de valores de dados inter-relacionados.

Nesse contexto, esses conjuntos de valores podem estar representando uma instância de uma entidade ou relacionamento da aplicação. Os nomes fornecidos às tabelas e às suas colunas podem auxiliar na compreensão do significado dos valores armazenados em cada uma das suas linhas. Em uma terminologia do modelo relacional, cada linha da relação é denominada de tupla, o nome da coluna é denominado atributo da relação, MCCARTY (1999).

O domínio consiste de um grupo de valores atômicos nos quais um ou mais atributo (ou colunas) retiram seus valores reais. O esquema de uma relação consiste de um conjunto de atributos que descrevem as características dos elementos a serem modelados. Os domínios nos quais os atributos da relação retiram seus valores não precisam ser necessariamente distintos, THOMPSON (2002).

Como esquema de um banco de dados relacional entende-se o conjunto de intenções definidas para todas as relações das informações, em um conjunto de restrições de integridade. Sobre os nomes fornecidos aos atributos, é permitido àqueles que representam conceitos semelhantes, possuir ou não o mesmo nome em diferentes relações. Da mesma forma, atributos representando conceitos diferentes podem possuir o mesmo nome. O conjunto de restrições de integridade define regras básicas que os

valores dos atributos devem obedecer quando aparecerem em uma relação, THOMPSON (2002).

2.3.2.2. Banco de Dados Objeto Relacional

O banco de dados objeto relacional é uma tecnologia com um modelo de dados relacional estendido, visando ser mais representativo em semântica e construções de modelagens. Estes SGBDs surgiram da necessidade das aplicações em utilizar certas características orientadas a objetos, sem no entanto, desconsiderar os grandes investimentos que foram feitos nos SGBDs relacionais.

Relacional estendido e objeto relacional são sinônimos para produtos que tentam unificar aspectos de modelo relacional e do orientado a objetos. Como uma tecnologia evolucionária, a abordagem objeto relacional tem herdado as transações robustas e as características de gerenciamento de performance do relacional e a flexibilidade do orientado a objetos, NASSU (1999).

Uma das principais características é permitir que o usuário defina tipos adicionais de dados, especificando a estrutura e a forma de operá-lo. Assim, os dados são armazenados em sua forma natural. Além dos tipos-base já existentes, é permitido ao usuário criar novos tipos e trabalhar com dados complexos como imagens, som e vídeo. Outra característica importante de um BDOR é vista na utilização de herança, funções e operações definidas pelos usuários.

Esta nova classe de banco de dados combina a velocidade e a capacidade de crescimento de um modelo multidimensional transacional com o poder e a flexibilidade da tecnologia de objetos. Devido à suas características únicas, os bancos de dados objeto relacionais são ideais para o desenvolvimento de aplicações de processamento de transações de elevado desempenho, NASSU (1999).

Os modelos multidimensionais facilitam bastante a modelagem dos dados, pois permitem representar estruturas reais complexas sem ignorar aspectos do mundo real e

sem ter que forçar esses aspectos a assumirem uma forma que possa ser digerida pela tecnologia. Além disso, há consideráveis vantagens durante a execução de processamentos complexos, NASSU (1999).

2.3.2.3. Banco de Dados Orientado a Objetos

O modelo de Banco de Dados Orientado a Objetos é uma tecnologia que integra banco de dados e a tecnologia de orientação a objetos. Aplicações de linguagens orientadas a objeto e sistemas estão exigindo capacidades de banco de dados, tais como continuidade, simultaneidade e transações nos seus ambientes, KHOSHAFIAN (1994).

Através de construções orientadas a objetos, os usuários podem encapsular os detalhes da implementação de seus módulos, compartilhar a referência a objetos e expandir seus sistemas através de módulos existentes. A funcionalidade de banco de dados é necessária para assegurar o compartilhamento concomitante e a continuidade das informações nas aplicações. Através dos bancos de dados, o usuário pode obter o estado em que os objetos se encontram e estar atualizando entre as várias solicitações de um programa. Vários usuários podem ao mesmo tempo compartilhar as informações, KHOSHAFIAN (1994).

A orientação a objetos é definida como :

Orientação a Objeto	=	*tipos de dados abstratos*	+
		Herança	+
		Identidade do objeto	

Processos de bancos de dados são definidas assim :

Processos bancos de dados	=	*continuidade*	+
		concomitância	+
		transações	+
		recuperação	+
		filtragem	+
		atualização	+

integridade +

segurança +

desempenho

Bancos de dados orientado a objeto são, portanto, definidos como segue :

Bancos de Dados Orientado a Objetos = *Orientação a Objetos* +
Aptidões de Bancos de Dados

Os bancos de dados orientados a objeto eliminam o assim chamado *hiato semântico* entre o campo da ação de uma aplicação e sua representação no armazenamento consistente. Desde que o mundo real seja modelado corretamente, as ligações e relações entre suas entidades são representadas e manipuladas diretamente. Os bancos de dados orientados a objetos propõem esta habilidade através dos conceitos orientados a objeto dos tipos de dados abstratos, herança e identidade do objeto, KHOSHAFIAN (1994).

Os bancos de dados orientados a objeto também reduzem a impedância da mistura entre as linguagens de programação e os sistemas de gerenciamento de banco de dados. Em aplicações complexas, o dado é recuperado de um sistema gerenciador de bancos de dados usando uma linguagem de filtragem de banco de dados e é então manipulado por rotinas desenvolvidas em uma linguagem de programação, KHOSHAFIAN (1994).

2.4. Objetos Persistentes

A orientação a objetos vem tornando-se rapidamente o modelo de programação de novas aplicações. Desde então, muitos programas necessitam tratar com objetos persistentes, NASSU (1999). A própria persistência pode ser definida como sendo os objetos que sobrevivem aos programas responsáveis por sua criação.

2.5. Finalidade da Persistência

Existem três importantes finalidades para o serviço de persistência de objeto :

- Suporte centralizado, constituído sob uma única interface apropriada para armazenamento de objetos em bancos de dados relacionais, bancos de dados objetos relacionais, bancos de dados orientado a objetos ou em sistemas de arquivos;
- Independência de banco de dados, onde utilizará uma simples interface no cliente para armazenar objetos em diferentes banco de dados, utilizando um mecanismo simples para implementar o suporte a objetos ;
- Arquitetura aberta, onde irá permitir que novos produtos de banco de dados possa agregar e se comunicar com os objetos em qualquer momento da fase de desenvolvimento da aplicação.

2.5.1. Suporte para Armazenamento de Objetos

O serviço de persistência de objetos oferece uma solução simplificada para o armazenamento de objetos. Os bancos de dados orientados a objetos são produtos habilitados para o armazenamento de qualquer objeto, independendo da escolha do formato na sua implementação e utilização. Como apresentado na figura 2.1, pode-se obter casos em que os objetos podem ser armazenados somente em banco de dados orientado a objetos, SESSIONS (1996).

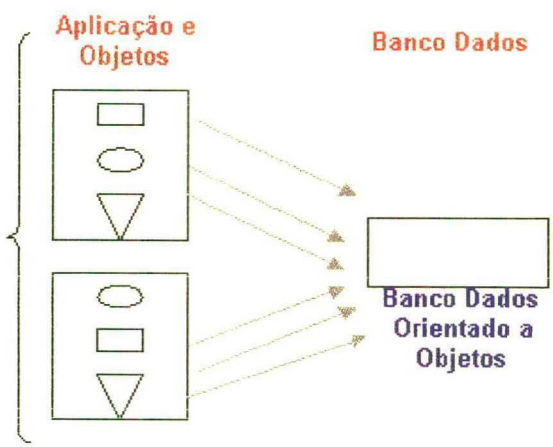

Figura 2.1- Situação necessária de armazenamento objetos, SESSIONS (1996)

Este fato é que em diferentes empresas a realidade é muito diferente da visão projetada anteriormente. Poucas das empresas atualmente estão habilitadas para o armazenamento

de informações em bancos de dados orientado a objetos. Atualmente, as empresas possuem e utilizam diversos produtos de armazenamento de informações como os descritos na figura 2.2. O serviço oferece uma interface para o armazenamento de objetos para as diferentes bases de dados tais como arquivos de dados, banco de dados relacional, base de dados objeto relacional.

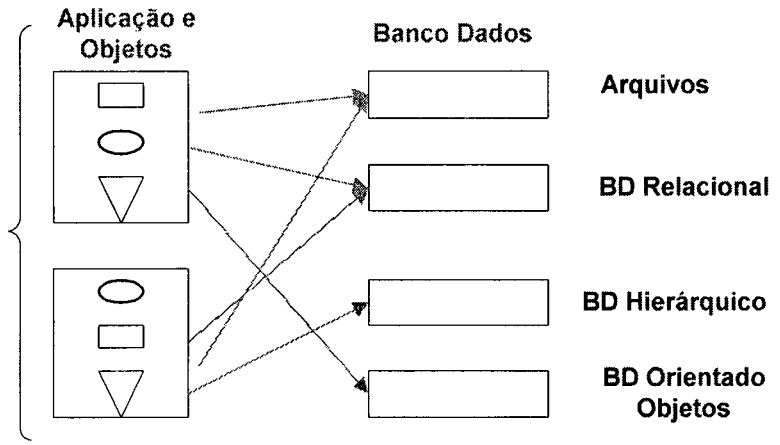

Figura 2.2- Situação de Armazenamento de Informações, SESSIONS (1996)

A visão do serviço de persistência de objetos é mostrada na figura 2.3.

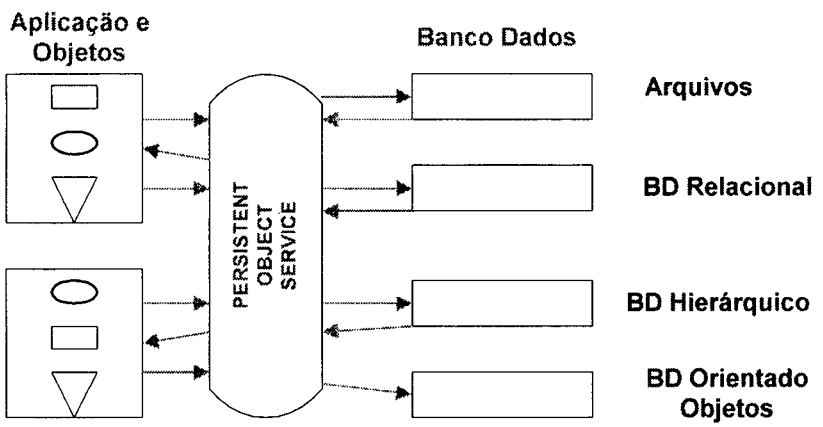

Figura 2.3- Arquitetura do Serviço Persistência, SESSIONS (1996)

Com o crescente e rápido interesse das corporações no desenvolvimento de aplicações utilizando linguagens orientadas a objetos, não ocorre o mesmo para os bancos de dados orientado a objeto pois a razão é que no mercado e nas grandes empresas se encontram as maiorias das informações armazenadas em banco de dados tradicionais e a maioria dos aplicativos foram desenvolvidos antes desta especificação.

2.5.2. Independência de Bancos de Dados

A independência de bancos de dados das aplicações para os clientes, está diferenciada para as aplicações e interfaces para o usuário final. Os códigos das interfaces dos usuários utilizam objetos que podem suportar o serviço de persistência de objetos, em grande escala, independente do banco de dados que está armazenando as informações, SESSIONS (1996).

Atualmente é possível desenvolver aplicações para cliente que estão em total independência com o banco de dados de armazenamento, onde esta é uma importante finalidade do serviço de persistência de objeto. Isso implica dizer que a mais importante finalidade da independência de banco de dados para os clientes é manter o banco de dados independente, mas contendo o mesmo nível do objeto. Entretanto, a independência de bancos de dados das aplicações servidoras, se torna mais crítica que as aplicações cliente, SESSIONS (1996).

Assim a finalidade da implementação deste serviço de objetos está em habilitar as suas classes e objetos para uma possível requisição da interface cliente, onde, se possuir a necessidade de adicionar novos códigos para a independência de bancos de dados, isso será possível desenvolvendo poucas linhas de códigos.

2.5.3. Arquitetura Aberta

O serviço de persistência de objeto pode ser considerado para a integração e dependência dos objetos entre dois sistemas imaginários, SESSIONS (1996). Este serviço inclui duas conexões ou conjunto de interfaces, como mostra a figura 2.4.

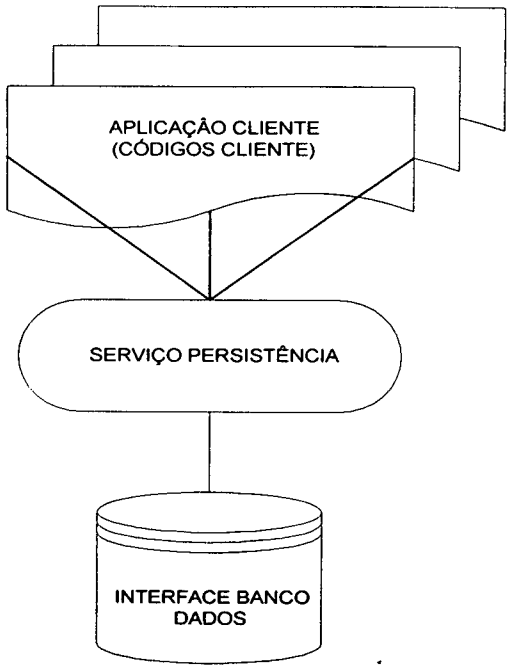

Figura 2.4- Níveis da Interface do Serviço de Persistência, SESSIONS (1996)

A primeira interface utilizada pelo código da aplicação do cliente é a interface que garante a independência de banco de dados. A segunda é o conjunto de interfaces que conecta com os diferentes bancos de dados. Porém, é esta interface que garante a arquitetura aberta, que proporciona todas as conexões da aplicação com os bancos de dados.

2.6. Abordagens para Implementação de Persistência

Numa perspectiva metodológica, verifica-se que existem três abordagens para implementar o serviço de persistência em programação orientada a objetos :

- **GOP** (*Gateway-based Object Persistence*): implica em adicionar em programação orientada a objetos, acessos para armazenar dados persistentes usando o tradicional armazenamento de dados não orientado a objeto. É usado em casos onde o usuário quer escrever aplicações usando modelos de programação orientada a objetos, onde o armazenamento de dados não é feito através da orientação a objetos. Um exemplo é a armazenagem de objetos em arquivos comuns.

- **ORDBMS** *(Object-relational DBMS)*: são construídos com a premissa de que estender o modelo relacional é o melhor caminho para novas aplicações orientadas a objetos. Levando-se em consideração que os modelos relacionais estão sendo bem sucedidos na prática e o SQL considerado um padrão global, o ORDBMS adiciona suportes para modelagem de dados orientada a objetos para estender ambos modelos de dados relacionais e linguagem de consulta, enquanto que as características da tecnologia dos DBMSs relacionais ficam relativamente intactas.

- **OODBMS** *(Object-oriented DBMS)*: são construídos com o princípio de que o melhor caminho para adicionar persistência de objetos é fazer objetos persistentes que são utilizados em linguagens orientadas a objetos. Pelo fato dos OODBMS terem suas origens nas linguagens orientadas a objetos, eles são freqüentemente considerados como sistemas persistentes de linguagens de programação. De qualquer forma, OODBMS vai muito além de simples adição de persistência em qualquer uma das linguagens de programação orientadas a objetos. OODBMS suporta avanços em aplicações de banco de dados orientados a objetos com características como suporte para objetos persistentes em mais de uma linguagem de programação, distribuição de dados, versões, generalização dinâmica de novos tipos, entre outras.

2.7. Estratégia de Persistência

As abordagens acima descritas, cada qual com suas características específicas, serviram de caminho para orientar a temática deste trabalho.

Nas literaturas utilizadas, detecta-se a existência de um padrão de armazenamento de objetos que foi desenvolvido sobre padrões existentes de linguagens de programação, objetos e bancos de dados com o objetivo de simplificar o armazenamento de objetos e garantir a portabilidade das aplicações. Este padrão estende Java, C++ e Smaltalk com facilidades completas para programação de bancos de dados, de tal forma que os desenvolvedores podem trabalhar apenas dentro do ambiente nativo da linguagem.

Pode-se armazenar objetos diretamente em banco de dados relacionais, objeto-relacional e orientado a objetos, que sejam compatíveis com o este padrão, SESSIONS (1996).

Com o propósito de facilitar o gerenciamento de mudanças, além de reduzir o tempo de desenvolvimento e a complexidade das aplicações, são mostradas as estratégias de persistência mais comuns:

- **Persistência por Herança:** Uma classe herda as capacidades de persistência de uma classe persistente pré-definida. O objeto instanciado desta classe pode ser persistente ou transitório, mas a operação de manipulação de dados entre a área persistente e transitória deve ser feita explicitamente.
  Esta estratégia não suporta persistência ortogonal (a identificação dos objetos persistentes é relatada para o tipo do sistema) e ainda asseguram um desequilíbrio da impedância.
- **Persistência por Instanciamento:** Um objeto é feito persistente e torna suas capacidades persistentes quando instanciado. Evidentemente esta estratégia não suporta persistência ortogonal (elemento que permite ou auxilia um objeto persistente ser explícito). Ocorre um problema nesta estratégia que quando as referências de um objeto persistente são transitórias. Este problema pode colocar a integridade referencial em desequilíbrio e é freqüentemente resolvido pelo uso de relacionamentos inversos. Usando este tipo de relacionamento, as estruturas das classes e implementações são mais rígidos, desta maneira um desequilíbrio de relacionamento é criado.
- **Persistência por alcançabilidade:** um objeto é feito persistente quando ele é alcançável de outro objeto persistente. Esta é a única estratégia de persistência que assegura a promessa de remover totalmente o desequilíbrio do relacionamento enquanto suporta persistência ortogonal, SESSIONS (1996).

2.8. Persistência para Aplicação JAVA

A persistência em aplicações Java pode ser obtida através de diferentes formas. As tecnologias existentes para alcançar persistência podem ser divididas em dois grupos

principais: as que propõem extensões da linguagem e da Máquina Virtual de Java para obter persistência ortogonal, como Pjama, e as que definem uma camada de serviço de persistência no topo da linguagem e utilizam os mecanismos de persistência existentes, como banco de dados relacional (BDR), onde a camada de serviços utilizada é composta pela API JDBC, REESE (2001)

JDBC é uma API que permite ao desenvolvedor Java usar a linguagem para obter acesso a uma grande quantidade de SGBDs e a outras fontes. JDBC foi projetado para ser simples, disponibilizando todas as tarefas do banco de dados de forma fácil, LINS (2000).

De forma simplificada, JDBC permite estabelecer a conexão com o banco de dados, enviar comandos SQL e processar os resultados. JDBC é uma interface de baixo nível, já que é utilizada para enviar diretamente comandos SQL, por isso serve também como base para construção de interfaces e ferramentas de mais alto nível e que fazem o mapeamento de classes para estruturas de bancos de dados, como tabelas do BDR.

Como apresentado na figura 2.5 a arquitetura JDBC é composto por três componentes principais: a API contendo classes e interfaces que definem os serviços a serem implementados pelos *drivers*; o *DriverManager* responsável pela gerência do conjunto dos *drivers* disponíveis à aplicação; e os *drivers* JDBC, os quais implementam as funcionalidades definidas nas interfaces da API. Estes *drivers* realizam as chamadas específicas do SGBD utilizado e se dividem em quatro tipos distintos:

1 (*JDBC-ODBC Bridge*) que se utilizam de *drivers* ODBC (*Open DataBase Conectivity*) já existentes;
2 (*Driver parcial*) que mapeia as chamadas JDBC para um API nativa;
3 (*Drivers Middleware*) que é um *driver* puro Java para um servidor *middleware* que suporta a utilização de aplicações clientes JDBC;
4 (*Direct-to-database*) que é um *driver* puro Java que permite a conexão com um servidor de banco de dados.

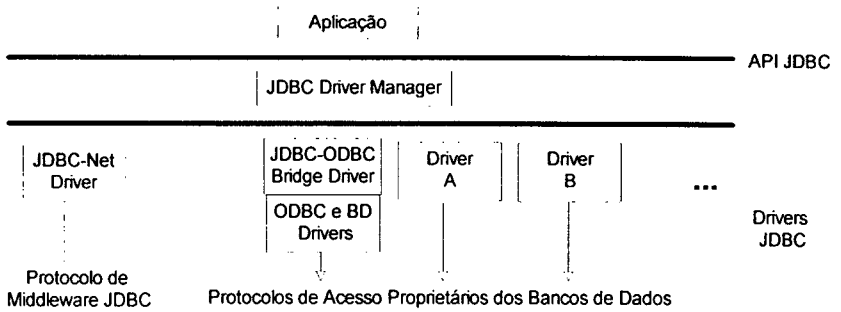

Figura 2.5- Arquitetura JDBC (Sistemas Persistentes), LINS (2000)

2.8.1. Formas de Persistência em JAVA

Os objetos Java possuem um ciclo de vida. Um objeto começa sua vida quando é criado, depois de criado o objeto existe até que seja destruído pelo coletor de lixo da máquina virtual Java, LINS (2000).

Persistência normalmente é implementada preservando o estado (atributos) de um objeto entre execuções do programa. Para preservar o estado, o objeto é convertido em um *stream* de *bytes* e usando algum tipo de armazenamento a longo prazo (normalmente um disco). Quando o objeto é requerido, é restabelecido do disco a longo prazo, o processo de restauração cria um objeto Java novo, idêntico ao original. Embora o objeto restabelecido não seja o mesmo objeto, seu estado e comportamento são idênticos, REESE (2001).

Há várias formas de persistência disponíveis para programadores em Java. As formas apresentadas abaixo incluem persistência baseada em arquivos, banco de dados relacional e banco de dados orientado a objetos. Estas formas de persistência diferem em várias categorias, incluindo: organização lógica do estado de um objeto, a quantidade de trabalho requerida para programar aplicações que suportem persistência, acesso simultâneo para o objeto persistente (de processos diferentes) e suporte a transações *commit* e semânticas *rollback*.

a) Arquivos

Dados armazenados em um arquivo podem ser simples (um arquivo de texto), ou podem ser complexos (um diagrama de circuito). Interage-se freqüentemente com objetos que são armazenados em arquivos (processamento de documentos de textos, planilhas eletrônicas, e assim por diante).

Arquivos podem ser usados como a base para um esquema de persistência em Java. Embora Java não suporte um mecanismo embutido para armazenar objetos em arquivos, Java provê uma biblioteca de fluxo portátil (*Data Input* e *Data Output*). Esta biblioteca torna mais fácil para o programador salvar e recuperar objetos.

Um mecanismo de persistência baseado em arquivo requer esforços do programador para alcançar persistência. O programador tem que escolher uma representação externa do objeto e desenvolver o código que salva e recupera os objetos. Objetos armazenados em arquivos são normalmente apropriados para aplicações de único-usuário que seguem o modelo Abrir/Fechar .. Arquivo/Salvar.

b) RDBMS (Banco de Dados Relacional)

Sistemas de Gerência de Bancos de Dados Relacionais (RDBMS) também podem armazenar objetos persistentes, mas as características de um banco de dados relacional são diferentes da persistência baseada em arquivos. Um banco de dados relacional é organizado em tabelas, linhas e colunas, em lugar de uma seqüência de *bytes* dentro de um arquivo.

Há dois modos principais para armazenar objetos em um banco de dados relacional. A primeira opção é interagir com o banco de dados em suas condições. A API JDBC provê interfaces que diretamente representam estruturas de banco de dados relacionais. Estas estruturas podem ser usadas e podem ser manipuladas. A outra opção é escrever sua própria classe Java e mapear entre as estruturas de dados relacionais e suas classes.

Ao usar um banco de dados relacional, que se configura como uma ferramenta para executar mapeamento para classes, deve-se escrever um volume grande de código para interagir com o banco de dados. Gerenciamento de objetos no banco de dados requer que sejam escritas declarações de SQL (*inserts, deletes, updates*, etc), que são remetidas ao banco de dados pela API JDBC.

Bancos de Dados Relacionais normalmente suportam controle de concorrência, onde múltiplos usuários podem ter acesso ao banco de dados sem interferências mútuas, porque o banco de dados usa controle de acesso. Adicionalmente, quase todos os bancos de dados relacionais suportam propriedades ACID (atomicidade, concorrência, isolamento e durabilidade). Estas propriedades protegem a integridade dos dados assegurando que as transações são totalmente efetivadas ou voltam ao seu estado inicial.

c) ODBMS (Bancos de Dados Orientados a Objetos)

Sistemas Gerenciadores de Banco de Dados Orientados a Objetos (ODBMS) suportam a persistência de maneira diferente da persistência baseada em arquivo e dos bancos de dados relacionais. A filosofia atrás dos bancos de dados orientados a objeto é tornar o trabalho do programador mais simples. Banco de Dados Orientados a Objetos armazenam objetos, onde o programador não tem que escrever declarações de SQL ou métodos de empacotamento de objetos. Sendo que a interface do banco de dados orientado a objeto normalmente leva em consideração estes detalhes.

Normalmente Banco de Dados Orientados a Objetos suportam controle de concorrência e propriedades ACID, como os bancos de dados relacionais. Eles provêem acesso de concorrência para banco de dados, assim como o controle de transações *commit e rollback.*

2.8.2. Serialização de Objetos

Serialização de objetos significa simplesmente gravar os valores de todos os seus campos de dados em uma seqüência de *bytes*. Assim que o objeto é transformado em

uma seqüência, ele pode ser gravado em um arquivo, que lhe proporcionará um objeto persistente que continuará existindo mesmo após o término do programa no qual foi criado, REESE (2001).

O processo de serialização de objetos permite converter a representação de um objeto em memória para uma seqüência de bytes que pode então ser enviada para um *ObjectOutputStream*, que por sua vez pode estar associado a um arquivo em disco ou a uma conexão de rede.

A serialização de objetos está na classe *java.io*. Abaixo estão relacionadas algumas classes importantes usadas:

- Classes de Conectividade:
    1. *java.sql.DriverManager* - gerencia os *drivers* que são registrados com ela
    2. *java.sql.Driver* - gerencia um tipo específico de banco de dados. Esta classe é uma interface.
    3. *java.sql.Connection* - representa uma sessão com um banco de dados específico. As instruções SQL são executadas e os resultados são retornados para uma conexão individual. Esta classe é uma interface.

- Classes de processamento de Dados:
    1. *java.sql.Statement* - envia uma instrução SQL. Esta classe é uma interface.
    2. *java.sql.ResultSet* - retém o resultado da execução de uma instrução SQL. Esta classe é uma interface.

O método *registerValidation*, permite registrar objeto para ser validado. Validar um objeto significa verificar se ele ainda está relacionado com o que ele estava relacionado antes. Se um objeto apontar para outros objetos, aqueles objetos também são salvos.

A palavra chave *transient* marca os dados não tendo valores salvos quando a classe é serializada. Um exemplo seria qualquer coisa, cujo valor seja somente válido em um determinado momento, como por exemplo: *transient int velocidade_atual.*

O serializador sabe que ele tem que preservar espaço para o valor, mas não salva um valor que está prestes a se tornar desatualizado. Por essa razão, os campos transientes também devem normalmente ser marcados como *private*, já que não se deseja que os outros objetos usem os dados que eles encontram lá, LINS (2000).

Com facilidades pode-se promover a serialização de uma classe, basta que ela implemente a interface *Serializable*. Entre as vantagens encontradas na serialização de objetos pode-se citar a portabilidade necessária para objetos remotos e o suporte à persistência. Nesse segundo caso, uma classe serializável poderá utilizar os recursos de banco de dados para uma aplicação, LINS (2000).

3. ARQUITETURA CORBA

A especificação CORBA (*Common Object Request Broker Architecture*) é um padrão para objetos que está sendo desenvolvido e proposto pela OMG (*Object Management Group),* que é uma organização internacional formada por muitas empresas membros, incluindo vendedores de sistemas de informações, desenvolvedores de *softwares* e usuários.

A OMG promove a aplicações da tecnológicas orientadas a objetos no desenvolvimento de sistemas distribuídos. O caráter da organização inclui o estabelecimento das especificações de gerenciamento de objetos para promover um padrão comum para desenvolvimento de aplicações. Os objetivos primordiais são: reusabilidade, portabilidade e interoperabilidade de *softwares* baseados em objetos; e em ambientes distribuídos e heterogêneos. Em concordância com tais especificações, será possível desenvolver um ambiente de aplicações heterogêneas usando as principais plataformas de *hardware* e sistemas operacionais.

O CORBA é uma arquitetura que permite aos objetos de sistemas distribuídos comunicarem-se entre si de forma transparente, não importando em qual plataforma ou sistema operacional estejam sendo executados, em que linguagem de programação foram implementados e até mesmo qual o protocolo de comunicação que eles utilizam, OTTE (1996).

Este modelo baseado em objetos permite que métodos de objetos sejam ativados remotamente, através de um elemento intermediário chamado ORB (*Object Request Broker*) situado entre o objeto propriamente dito e o sistema operacional, acrescido de funcionalidades que o permitam comunicar-se através da rede.

O ORB é o principal componente da arquitetura, que provê os mecanismos de comunicação entre os objetos. O ORB fornece uma estrutura que permite objetos conversarem entre si, independente de aspectos específicos da plataforma e técnicas usadas para implementá-los, ou seja, um cliente pode invocar, transparentemente, um

método num objeto servidor, o qual pode estar na mesma máquina ou em qualquer lugar da rede. O ORB intercepta a requisição e fica responsável por encontrar um objeto que implemente a operação requerida, passar os parâmetros da invocação ao objeto invocado e retornar o resultado ao cliente. O cliente não precisa saber onde o objeto está alocado, não precisa saber qual a linguagem que o implementa, seu sistema operacional ou qualquer outro aspecto do sistema que não seja relacionado com a interface do objeto, ORFALI (1997).

A invocação de métodos via ORB pode ser de dois tipos:

- **estática** - o nome de um método é fornecido em tempo de compilação, nesse caso o cliente tem conhecimento prévio de qual servidor atenderá a sua solicitação;
- **dinâmica** - o método que será executado é definido somente em tempo de execução. O cliente não tem conhecimento sobre qual objeto oferece um serviço ou como está implementada sua interface. Esse recurso é muito importante para a implementação de mecanismos de tolerância a falhas e de chamadas remotas eficientes onde não se sabe qual servidor atenderá a solicitação do cliente.

Este trabalho prioriza na sua estrutura a invocação estática de métodos, por considerar este método como o mais apropriado.

A arquitetura deste padrão possibilita o uso de mecanismos pelos quais um objeto pode enviar requisições ou receber respostas, para outro objeto no sistema distribuído, de uma forma transparente como definido pelo ORB do OMG. O ORB sob tal arquitetura é uma aplicação que possibilita interoperabilidade entre objetos, construídos em linguagens diferentes, executados em diferentes máquinas em ambientes heterogêneos distribuídos.

Aplicações típicas cliente/servidor usam seus próprios *design* ou um padrão reconhecido para definir o protocolo a ser usado entre os dispositivos. Definições de protocolos dependem da linguagem de implementação, transporte na rede entre outros fatores. Os ORBs simplificam este processo. Com um ORB, o protocolo de rede é abstraído, o usuário se preocupa apenas com a interface com os objetos, como se estes fossem locais. Os usuários podem definir o protocolo de comunicação entre seus objetos,

durante a especificação de suas interfaces. Seguir os padrões da arquitetura CORBA garante portabilidade e interoperabilidade de objetos sobre uma rede de sistemas heterogêneos, ORFALI (1997).

Para mostrar a estrutura OMA, a figura 3.1 contém os principais componentes da arquitetura CORBA: corretor de requisição de objetos (ORB), serviços comuns de objetos , facilidades comuns e aplicações de objetos.

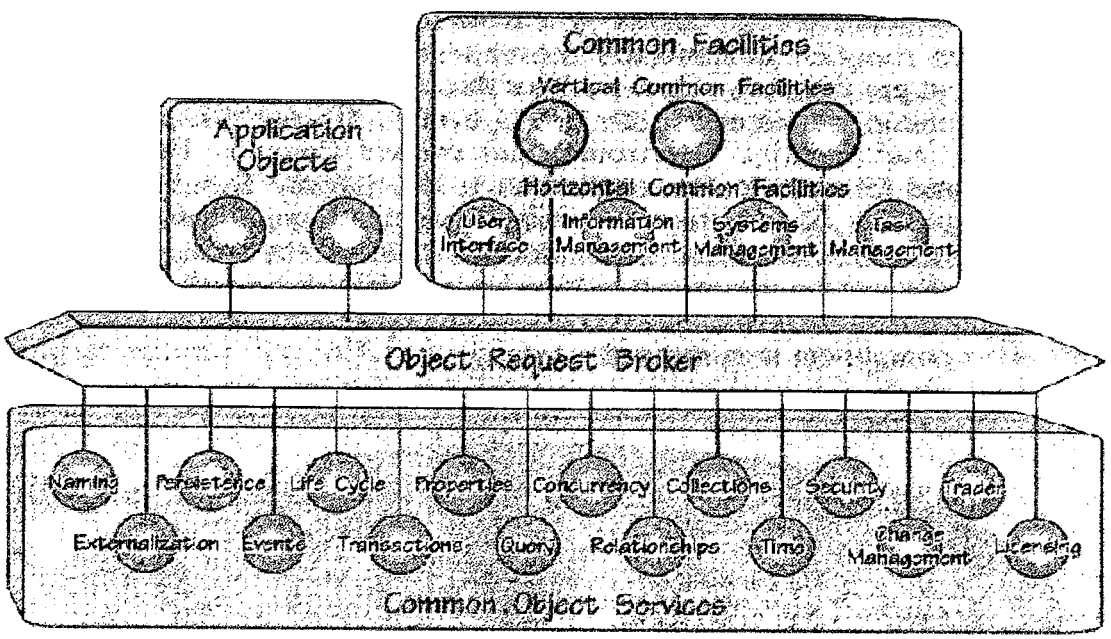

Figura 3.1- Estrutura do OMA, ORFALI (1997)

O ORB é uma camada intermediária que estabelece o relacionamento cliente/servidor entre os objetos. Usando um ORB, um objeto cliente pode invocar de forma transparente um método de um objeto servidor, o qual pode estar na mesma máquina ou em um lugar qualquer da rede, ORFALI (1997).

O OMA (*Object Management Architecture*). e seus principais componentes são:

- Núcleo CORBA e ORB - manipulam requisições entre objetos;
- Serviços CORBA - definem serviços a nível de sistema que ajudam a gerenciar e manter objetos;

- Facilidades CORBA - definem facilidades e interfaces no nível de aplicação - manipulação de dados e armazenamento;
- Objetos de Aplicações - são os objetos propriamente ditos no nível visível de aplicação.

O ORB é o componente mais importante da arquitetura OMA. Ele permite que objetos façam e recebam requisições de métodos transparentemente em um ambiente distribuído e heterogêneo. O CORBA é uma especificação das interfaces do ORB, ou melhor, um modelo concreto para a especificação abstrata dada no documento que define a arquitetura OMA.

Os *Commom Object Services* são coleções de serviços executados dentro de um sistema e agrupados em forma de componentes. Eles complementam as funcionalidades do ORB. São utilizados para criar um objeto, nomeá-lo e torná-lo disponível no ambiente. Atualmente, a OMG possui 16 serviços de uso comum especificados. Nota-se que nesta parte dessa arquitetura está situada o serviço de persistência do CORBA.

As *Common Facilities* são coleções de componentes que oferecem serviços diretamente aos objetos da aplicação. A diferença marcante entre *services* e *facilities* está no nível em que cada um se aplica: os *services* estão intimamente relacionados ao ORB, enquanto as *facilities* estão intimamente relacionadas aos objetos da aplicação, ORFALI (1997).

3.1. Utilização da Padronização CORBA

Os objetos oferecem um modelo natural para sistemas distribuídos abertos porque os componentes desses sistemas só podem se comunicar usando mensagens, através de uma interface bem definida, adequando-se às características de desacoplamento e extensibilidade. A interoperabilidade entre objetos distribuídos pode ser garantida através da adoção de padrões, como a arquitetura CORBA, OTTE (1996).

A proposta elaborada pelo OMG permite a comunicação entre objetos em ambientes distribuídos e heterogêneos. Normalmente, as propostas para interoperabilidade procuram definir um modelo integrador através da padronização de seu modelo, de suas APIs e de seu protocolo, facilitando o desenvolvimento independente de componentes heterogêneos e distribuídos. Elas adotam uma linguagem de definição de interface IDL (*Interface Definition Language*), estritamente declarativa, para descrever tipos de objetos que passam por um compilador que traduz para adaptadores, chamados de *stubs*, em uma linguagem especifica. Os *stubs* gerados podem ser assim ligados com os módulos clientes e servidores para compor a aplicação final. É com essa solução que a OMG define o padrão aberto CORBA, cujo objetivo é a integração de aplicações, resolvendo o problema da interoperabilidade através da arquitetura CORBA.

Um objeto CORBA é uma entidade abstrata que representa um conceito concreto do mundo real. É formado por dados e métodos que podem ser chamados por outros objetos CORBA, pois oferece um conjunto de atributos e operações que compõem a sua interface IDL. A função da IDL é tornar transparente a implementação do objeto. Um cliente do objeto simplesmente usa essa interface para chamar os métodos, enfatizando a transparência de localidade, plataforma ou detalhes de implementação, OTTE (1996).

Nesse ambiente, os componentes distribuídos podem assumir dois papéis: de cliente e/ou de servidor. No papel de cliente, um componente envia requisições de operações para um objeto servidor que disponibiliza a respectiva implementação da interface IDL chamada. Para que um componente cliente possa fazer uma chamada a um outro, é necessário apenas que conheça a interface IDL e a referência de objetos interoperáveis do objeto CORBA servidor. Do lado do cliente descreve-se os seguintes itens:

- *Stubs* – responsáveis pela invocação de um cliente a um objeto remoto de forma transparente, encapsulando as chamadas primitivas de sistemas de entrada/saída.
- Interface de Invocação Dinâmica (DII) é responsável por permitir que uma invocação a um objeto servidor seja apresentada dinamicamente, por um cliente em tempo de execução.

- Repositório de Interfaces é um serviço que fornece objetos persistentes, representando as informações sobre a IDL. Tais informações podem ser consideradas metadados dos objetos CORBA.

Do lado do objeto CORBA servidor descreve-se os seguintes itens:

- ORB que permite o acesso às funcionalidades não oferecidas pelas demais interfaces, pois não depende da interface do objeto ou do adaptador do objeto.
- Adaptador de Objetos é responsável pelos serviços básicos de geração e interpretação de referência de objetos IOR e pela ativação e desativação de objetos CORBA e suas correspondentes implementações em servidores.
- *Skeletons* Estáticos são responsáveis por receber uma invocação remota a um objeto CORBA e enviar uma resposta de forma transparente.
- Esqueleto Dinâmico é responsável por oferecer um mecanismo de acesso a objetos em servidores que não tem os correspondentes esqueletos estáticos.
- Repositório de Implementação é responsável pelo registro das informações que permitam ao ORB localizar e ativar implementações de objetos CORBA.

Vê-se que uma vez conhecida a interface de um objeto servidor, um cliente CORBA pode ter acesso a seus métodos através das interfaces estáticas ou dinâmicas.

3.1.1. Vantagens e Desvantagens

Dentro das vantagens da arquitetura CORBA pode-se destacar:

- **A interface é independente da linguagem de programação:** Interfaces entre clientes e servidores são definidas em OMG IDL, desta forma fornecendo as seguintes vantagens para a programação *Internet*;
- **Independência de linguagem e ambiente de plataformas:** Separação bem definida entre interface e implementação, geração automática de código *stubs* através do compilador IDL, checagem de tipo automática.

- **Integração Legada (*Legacy integration*):** Utilizando CORBA IDL, programadores podem encapsular aplicações existentes em *wrappers* e usá-los como objetos sobre ORB. Sistemas baseados em CORBA permitem aos desenvolvedores reutilizar aplicações legadas portanto, prevenindo-os de desenvolver linhas de código;
- **Com toda a infra-estrutura de objetos distribuídos:** Aplicações distribuídas requerem mais funcionalidades do que simples métodos de invocações. CORBA oferece um conjunto de serviços e facilidade de objetos, como o serviço de ciclo de vida, serviço de evento, serviço de nomes e outros mais. CORBA fornece muitas outras facilidades para gerenciar aplicações distribuídas semelhantes a ativação do servidor automático, balanceamento de carga para distribuí-la através de múltiplas máquinas, e persistência para ambos estados de objetos e referências aos objetos. Alguns produtos Corba também implementam desativação automática do servidor;
- **Transparência de localização:** CORBA fornece transparência de localizações em uma referência a um objeto, onde é independente da localização física do objeto e provável mudança desta localização. Isto significa que desenvolvedores podem criar sistemas baseados em Corba, onde os objetos podem ser movidos sem que as aplicações clientes se preocupem;
- **Transparência em rede:** Por usar o protocolo IIOP, um ORB que roda em uma máquina particular que pode interconectar com qualquer ORB sobre a rede. Uma interface cliente gera as chamadas e receberá as respostas com os resultados esperados ou exceções, sem conhecer se a computação é feita no mesmo processo, em diferentes processos na mesma máquina ou em outra máquina com outro sistema operacional na rede;
- **Comunicação direta entre objetos:** Uma vez que os *applets java* são carregados no *WEB Browsers*, eles podem conversar diretamente com outros objetos distribuídos, utilizando assim o servidor HTTP. Esta transparência no roteamento da mensagem é feita através do IIOP e é livre de comunicação extra dos *WEB Servers*;
- **Interface de invocação dinâmica:** Clientes CORBA podem usar ambos, invocação de métodos dinamicamente e estaticamente. Estaticamente, os

métodos são invocados em tempo de compilação, caso contrário, dinamicamente, os métodos são descobertos em tempo de execução. Invocação estática oferece uma forte checagem de tipos e a dinâmica oferece uma flexibilidade na associação, OTTE (1996).

Já nas desvantagens da arquitetura CORBA cita-se:

- **Tempo de *Download* Extra:** Para estar apto a comunicar com outros objetos CORBA, o *applet* rodando em um *browser Web* necessita de um *Orb* para trocar informações. Portanto, um *Orb* é carregado com as classes que formam a *applet*. Estes *Orb* são apenas classes em java, mas ainda necessitam ser carregado a cada vez que um *applet* é executado;

- **Tecnologia Complexa:** Necessita-se de treinamento e investimento na nova arquitetura, o que é natural, quando se troca de métodos, metodologias e paradigmas;

3.2. Serviços e facilidades CORBA

A arquitetura CORBA fornece dezesseis serviços fundamentais para aplicações orientadas a objetos e seus componentes. Abaixo segue a descrição do serviço de persistência.

Persistent Object (Persistência): proporciona uma interface única para os objetos acessarem os vários mecanismos de persistência, bases de dados relacionais, bases de dados OO, e arquivos simples.

Os serviços de objeto COSS (*Common Object Services Specifications*) formam uma coleção de serviços de sistema (interfaces a objetos), que oferecem funções básicas para utilizar e implementar os objetos. O COSS pode ser entendido como uma extensão ou como uma complementação das funcionalidades do ORB. O COS (*Commom Object Services)* são coleções de serviços executados à base de sistemas e agrupados em forma

de componentes. Eles complementam a funcionalidade do ORB. São utilizados para criar um objeto, nomeá-lo e torná-lo disponível no ambiente.

O ORB é o *middleware* que estabelece o relacionamento cliente/servidor entre os objetos. Usando um ORB, um objeto cliente pode invocar de forma transparente um método de um objeto servidor, o qual pode estar na mesma estação ou em outro qualquer no ambiente, BROSE (2001).

Este serviço possui duas formas de armazenamento, para um armazenamento de único nível (*Single-Level*) a interface cliente não é informada se o objeto está na memória ou em disco. Em contraste, o armazenamento de dois níveis (*Two-Level*) separa memória de armazenamento persistente. O objeto deve ser explicitamente passado de um banco de dados (ou arquivo) para dentro da memória e vice-versa, BROSE (2001) .

3.3. Visão Geral da Arquitetura CORBA

A especificação do CORBA define como as aplicações cliente se comunicam com objetos implementados em um servidor. Essa comunicação é feita através de um *Object Request Broker* (ORB). O ORB provê os mecanismos de comunicação entre os objetos, fornecendo uma estrutura que permite aos objetos conversarem entre si, independentemente de aspectos específicos da plataforma e das técnicas usadas para implementá-los, ou seja, um cliente pode invocar, transparentemente, um método num objeto servidor, o qual pode estar na mesma máquina ou em qualquer lugar da rede, OTTE (1996).

O ORB intercepta o requerimento invocado e fica responsável por encontrar um objeto que implementará a operação requerida, por passar os parâmetros da invocação ao objeto invocado e por retornar o resultado ao cliente. O cliente não precisa saber onde o objeto está alocado, não precisa saber qual a linguagem que o implementa, qual o seu sistema operacional ou qualquer outro aspecto que não seja relacionado com a interface do objeto.

Aplicações típicas cliente/servidor usam seus próprios *designs* ou um padrão reconhecido para definir o protocolo a ser usado entre os dispositivos. Definições de protocolos dependem da linguagem de implementação, transporte na rede e de vários outros fatores. Os ORBs simplificam este processo. Com um ORB, o protocolo de rede é abstraído, o usuário se preocupa apenas com a interface com os objetos, como se estes fossem locais. Portanto, seguir os padrões da arquitetura CORBA garante portabilidade e interoperabilidade de objetos sobre uma rede de sistemas heterogêneos.

O ORB provê as seguintes funcionalidades:

Nomeação: cada conjunto de nós participantes do escopo do ORB possui um esquema básico de nomeação que permite que, uma requisição de um objeto seja mapeada no objeto que implementa o serviço.

Despacho da requisição: o ORB providencia o despacho da requisição de um serviço para o servidor adequado, lidando com os aspectos específicos da linguagem de interfaceamento.

Serialização: o ORB é responsável pela serialização dos parâmetros das chamadas de serviços, envio pela rede e pelo protocolo de rede e des-serialização na máquina de destino.

Os serviços que os objetos oferecem são especificados através de uma linguagem de especificação de diálogos a IDL, que possibilita que os serviços sejam representados de forma a que possam ser usados por todos os objetos que assim o desejem, independentemente da linguagem de programação desses objetos. Os clientes acessam os serviços usando um *stub* gerado automaticamente ou usando um serviço de invocação dinâmica, OTTE (1996). A figura 3.2 mostra uma requisição sendo enviado por um cliente a uma implementação de objeto.

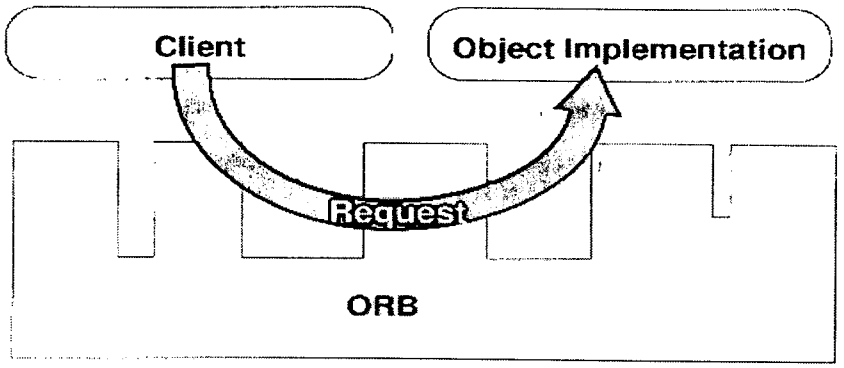

Figura 3.2- Requisição do Cliente para uma Implementação, OTTE (1996)

O ORB é responsável por todo o relacionamento com o mecanismo necessário para encontrar o objeto da requisição, para preparar o objeto para receber a requisição e comunicar os dados. Vê-se que a interface que o cliente utiliza é completamente independente de onde o objeto está alocado, de qual linguagem implementa-o ou de qualquer outro aspecto que não esteja relacionado com a interface em questão.

Tudo que o cliente pode utilizar é composto por interfaces. Para o caso de uma invocação a uma operação por uma forma dinâmica, o cliente irá tratar com a interface de invocação dinâmica. Neste caso, independentemente de qual seja a implementação de ORB usada, o cliente verá sempre a mesma interface, isto é, sempre aparece a mesma interface em todos os ORB para aquela dada requisição, justamente devido à arquitetura CORBA exportar suas interfaces públicas no padrão IDL. O cliente também pode usar o método de invocação não dinâmica. Neste caso, ele verá a interface com IDL *Stubs*, a qual contém dados específicos de interfaces conhecidas em tempo de compilação; para cada tipo de objeto haverá uma interface visualizada pelo cliente nos IDL *Stubs*. Na invocação dinâmica, há a possibilidade de utilizar objetos que necessitam de funcionalidades que estão em máquinas remotas. Logo, é feita uma amarração dinâmica entre esses objetos na hora de sua instanciação.

Como também está mostrada na figura 3.2, existem interfaces que são dependentes do ORB que está sendo usado. Entretanto, estas interfaces nunca são visualizadas pelo cliente, pois são de uso interno ao ORB. O cliente também pode interagir diretamente com o ORB, mas por uma interface específica para isto.

O objeto é implementado e, então, executa-se a operação requerida. A implementação do objeto recebe uma requisição como uma chamada proveniente de um esqueleto estático ou um dinâmico. Durante o desenvolvimento de uma operação, um objeto pode usar algum serviço do ORB usando o adaptador de objetos.

3.3.1. Repositório de Interface

O repositório de interface é um serviço que provê objetos persistentes (durante o tempo de vida do repositório de interfaces), que representam a informação IDL numa forma disponível em tempo de execução. A informação no repositório de interface pode ser usada pelo ORB para suprir requisições, OTTE (1996).

Usando a informação no repositório de interface, fica possível a um programa invocar operações em um objeto cuja interface não era conhecida em tempo de compilação, isto é, ser capaz de determinar que operações são válidas no objeto e fazer uma invocação nelas.

O repositório de implementação é o local onde são guardadas informações sobre implementações de objetos e dados.

3.3.2. A Estrutura de um Cliente

Um cliente tem uma referência a um objeto, no qual pode-se referir ao mesmo. O cliente pode, então, invocar operações nesta referência de objeto, como se a referência fosse o próprio objeto. Uma referência a um objeto é um *token* que pode ser invocado ou passado como parâmetro a uma invocação de um objeto diferente. Invocar um objeto envolve especificar o objeto a ser invocado, a operação a ser executada e os parâmetros a serem dados para a operação e/ou parâmetros retornados da invocação. No momento em que um ORB não pode completar uma invocação, uma exceção é levantada.

Uma referência a um objeto também pode ser convertida numa *string* que pode ser armazenada em qualquer arquivo ou comunicada por diferentes meios; e subseqüentemente transformada novamente em referência a um objeto pelo ORB que produziu a *string*. A substituição de uma referência por uma *string* é uma maneira muito flexível de representação, já que uma *string* pode ser comparada, copiada ou movida.

Nunca existirão dois objetos idênticos instanciados num sistema. Assim, cada objeto diferente deve ter sua própria referência criada pelo ORB. O ORB cria tais *strings* usando métodos especiais que levam em consideração dados como: IP da máquina (domínio), hora e data.

O objetivo desta consideração é criar *strings* sempre diferentes das já criadas, isto é, evitar conflitos de *strings*. Também, sob este propósito, as *strings* são criadas com um tamanho muito grande. Quanto maior a string menor é a probabilidade de conflitos.

3.3.3. Estrutura de uma Implementação de Objeto

Muitas das implementações de objetos provêem seu comportamento usando facilidades como o ORB e adaptadores de objeto. Por exemplo, o *Basic Object Adapter* (BOA) fornece algum dado persistente associado a um objeto. Esta quantidade de dados (relativamente pequena) é tipicamente usada para armazenar um identificador, OTTE (1996).

A implementação de objeto, como mostrada na figura 3.3, pode utilizar esse identificador de métodos para instanciar objetos persistentes armazenados em um serviço de armazenagem da escolha de uma implementação de objeto. Com esta estrutura, fica possível, não somente para implementações diferentes de objetos, usar o mesmo serviço de objeto, mas aos objetos, escolher o serviço que é mais apropriado a eles.

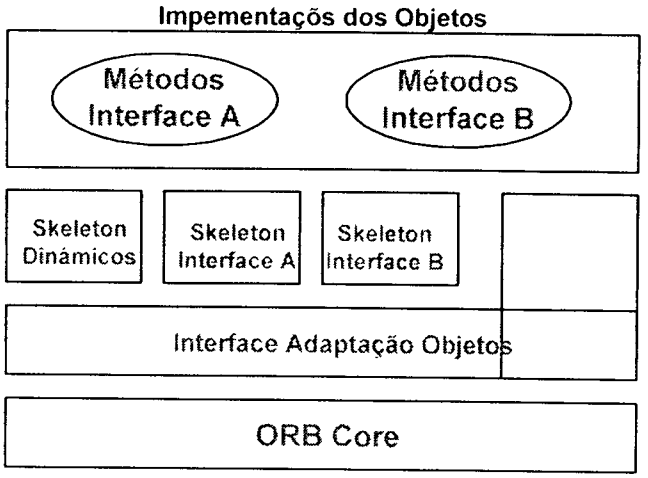

Figura 3.3- Estrutura de uma Implementação de Objeto, OTTE (1996)

Esses adaptadores são responsáveis pelas seguintes funções:

- Geração e interpretação de referências a objetos;
- Chamada de métodos;
- Ativação e desativação de objetos;
- Mapeamento de referências a objetos correspondentes;
- Registro de implementações.

Essas funções são desenvolvidas usando-se o ORB *Core*. O adaptador de objetos define a maioria dos serviços do ORB que são utilizados na implementação de objetos. Se uma implementação de objeto precisa de um dado valor no ato de sua invocação, esse valor poderá ser guardado juntamente com sua referência.

Dessa forma, por exemplo, ao usar a referência, o ORB também terá outro dado a respeito do objeto criado. Se o ORB não permitir esse tipo de coisa, então o adaptador de objetos poderá ser usado para guardar tal dado em seu próprio armazém de dados.

Com os adaptadores de objetos, fica possível para uma implementação de um objeto ter acesso a um serviço que é ou não implementado no ORB *Core*. Se o ORB *Core* provê este serviço, então o adaptador de objetos simplesmente provê uma interface para o mesmo, do contrário, o adaptador deve implementar o serviço no topo do ORB *Core*.

4. SERVIÇO DE PERSISTÊNCIA DO CORBA

O serviço de persistência do CORBA é padronizado pela OMG dentro da arquitetura OMA (*Object Management Architeture*).

A forma de armazenamento mais utilizado é o de dois níveis, e os objetos são armazenados em ODBMSs, os quais proporcionam o caminho mais direto para o armazenamento de objetos.

O PSS (*Persistent State Service*) é um serviço de persistência CORBA que surgiu para tornar mais transparente o acesso ao sistema de armazenamento. O PSS fornece uma interface única para armazenar o estado persistente dos objetos em uma variedade de dispositivos de armazenamento (BDR, BDOR, BDOO e arquivos). Trata-se de uma interface entre o CORBA (*servants, POAs* e outros) e os sistemas de armazenamento de dados que permite salvar e restabelecer o estado dos objetos de forma totalmente transparente para os clientes da aplicação, PEREIRA (2000).

4.1. PSS (Persistent State Service)

A OMG especificou o PSS (*Persistent State Service*) para efetuar o estado de transação com o estado persistente do objeto. É um serviço de desenvolvimento que tem a definição de meios para estados persistentes. Para as operações do cliente este serviço é totalmente transparente, PEREIRA (2000).

Aplicações realmente requerem persistência, de uma forma ou de outra. A referência de persistência e o estado da persistência não são necessariamente utilizados em conjunto, mas sempre poderão estar. Mas é possível, entretanto, possuir referencias persistentes do objeto com estado temporário ou igual a ter um objeto temporário que utiliza recursos do estado persistente, BROSE (2001).

Esta versão do serviço de persistência CORBA surgiu para tornar mais transparente o acesso ao sistema de armazenamento. O PSS fornece uma interface única para

armazenar o estado persistente dos objetos em uma variedade de dispositivos de armazenamento (BDR, BDOO e arquivos). A nova especificação para o serviço estabeleceu uma linguagem, a PSDL (*Persistent State Definition Language*), que possui construtores os quais definem objetos persistentes e dispositivos de armazenamentos.

Duas etapas são necessárias para empregar esta linguagem no processo de armazenamento dos objetos persistentes. A primeira consiste em criar, para cada classe persistente da aplicação, um arquivo com extensão PSDL contendo a descrição dos atributos que devem ser armazenados no banco de dados. A segunda, caracteriza-se pela compilação e geração dos códigos necessários para armazenar/recuperar os objetos persistentes da aplicação, PEREIRA (2000).

4.1.1. Concepção do PSS

PSS permite automatizar, tanto quanto possível, o mapeamento entre os objetos no servidor e no repositório (arquivos, bases de dados relacionais, bases de dados orientadas a objetos).

Toda a estrutura do banco de dados pode ser definida pela linguagem PSDL que são utilizadas pelas implementações para mapear as estruturas dos dados, mas isso não é requerido para o desenvolvimento, onde será utilizada sessão genérica para a estruturação e mapeamento dos dados.

Para a estrutura do serviço PSS, apresentada na figura 4.1 com todos os processos de integração, o PSS define a interface para o desenvolvimento de sessões estáticas e dinâmicas para mapeamento e gerenciamento da estrutura dos dados, BROSE (2001).

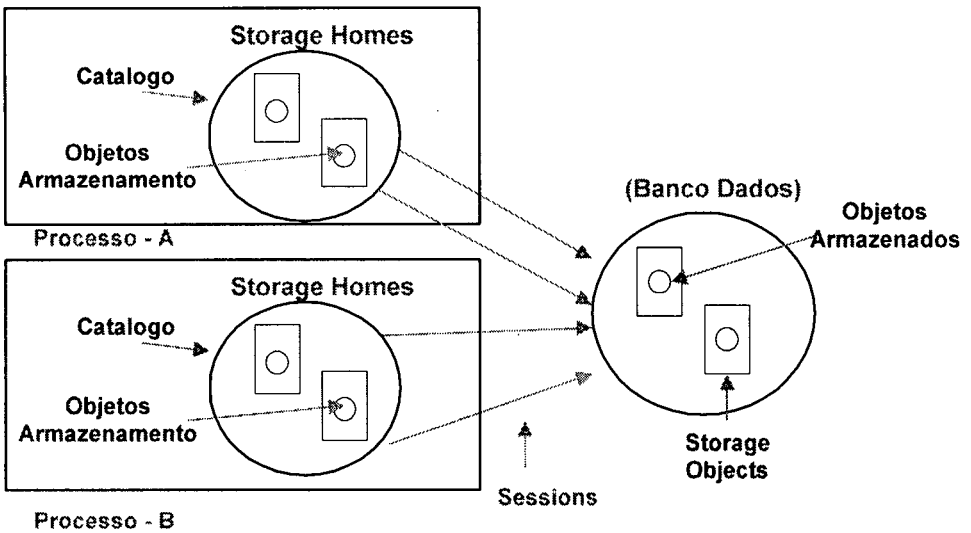

Figura 4.1 Estrutura do Serviço de PSS, BROSE (2001)

Principais conceitos:

- *Datastore*
  - é o mecanismo de armazenamento (BD, Arquivos).

- *Storage Homes*
  - são partições do *datastore* onde os *storage objects* são armazenados;
  - aparecem dentro do *datastore* e dentro de catálogos (no processo do servente);
  - possuem um tipo definido em PSDL.

- *Storage Objects* (objetos de armazenamento)
  - os dados em si, equivalentes a tuplas ou linhas em bancos de dados ;
  - possuem um tipo (*storagetype*) definido estaticamente em PSDL;

- *short-pid* define um objeto de armazenamento unicamente dentro de um *home;*
- *pid* define um objeto de armazenamento unicamente globalmente.

- *Catalog*
  - são conjuntos de storage objects.

Para o suporte do serviço de Persistência PSS poderá ser determinado por meios de armazenamentos, tem-se :

1. **Sistemas de Arquivos :**

- Não suporta dados complexos (implica conversão do formato dos dados explícito para cada tipo) EX : CORBA *Externalization*, Java *Serialization*, COM *Structured Storage*;
- Não suporta transações;
- Independente da linguagem de programação

2. **Base de Dados Relacional :**

- Facilidade de Utilização;
- Limitado a ambientes *single-vendor/single-server;*
- Interação apenas via RPC;
- Suporte de transações limitado;
- Não suporta dados complexos (implica conversão do formato dos dados explícito para cada tipo)

3. **Base de Dados Orientados a Objetos :**

- Facilidade de Utilização;
- Limitado a ambientes *single-vendor/single-server;*
- Suporta dados complexos;
- Dependente da linguagem de programação

O serviço PSS necessita de uma arquitetura de estrutura para suportar o processamento e realização dos serviços. Estes processos estão descritos na figura 4.2, onde realiza a interação de chamadas entre API e repositórios de dados. Todos gerando a arquitetura do serviço.

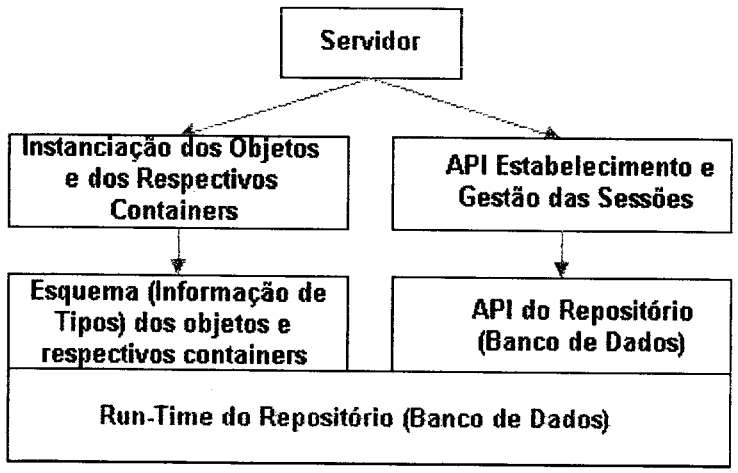

Figura 4.2- Arquitetura do Serviço de PSS, BROSE (2001)

4.1.2. Linguagem PSDL

O PSS oferece uma linguagem de definição de dados orientadas a objetos, a PSDL (*Persistent State Definition Language*) que é muito mais apropriada para persistir o estado do objeto em SQL. Em algumas situações a PSDL é designada para mapear banco de dados SQL, Objeto Relacional, Orientado a Objetos entre outros tipos de armazenamento, mas poderá implementar PSS utilizando a interface JDBC, BROSE (2001).

O PSS define então uma linguagem de concepção, apresentando uma API que executa o estado persistente do objeto. Tal API já existe em JAVA, que acessa banco de dados SQL utilizando a interface JDBC. Mas a diferença é que a PSDL é transparente e apropriada para persistir objetos que o SQL. Pode-se implementar PSS utilizando a interface JDBC onde observa-se implementações PSS com JAVA e JDBC, conectando com diversos banco de dados, BROSE (2001).

```
//Pessoas.psdl
abstract storagetype Pessoa {
```

```
   readonly state long RG;
   state string nome;
   state string telefone;
   state ref<Pessoa> conjuge;
   void casamenton (in ref<Pessoa> o_conjuge);
};

abstract storagehome PessoaHome of Pessoa {
   factory create(      in long RG, in string nome,
                        in string telefone);
};

catalog Pessoas {
  provides PessoaHome pessoa_home;
};
```

Figura 4.3- Estrutura da Linguagem PSDL

Para a utilização da especificação PSDL se faz necessário compiladores de fornecedores para implementações PSS, que podem relacionar-se com esquema de uma linguagem particular de banco de dados. Entretanto, as linguagens de programação deverão construir classes JAVA para a definição da abstração na linguagem PSDL. Assim sendo, para cada classe persistente da aplicação deve existir um arquivo PSDL. Alterações nestas classes, como por exemplo, a inclusão de um novo atributo sem que se altere o arquivo PSDL correspondente, podendo comprometer a integridade do armazenamento dos dados, BROSE (2001).

4.2. Implementação do PSS

O *Persistent State Service* (PSS) é um serviço diferente dos outros disponibilizados pela OMG, pois ele mapeia os estados de objetos desenvolvidos na implementação, tornando assim objetos persistentes. Este serviço define também uma nova linguagem de definição do estado persistente do objeto, o *Persistent State Definition Language* (PSDL). Esta linguagem é uma evolução da IDL onde permite definir detalhes do modelo de dados da implementação, criando assim um segundo nível de abstração entre as interfaces CORBA e da implementação desenvolvida. As definições do PSDL são interpretadas pelos compiladores do PSDL e mapeadas para o desenvolvimento de mecanismos de persistência da implementação, utilizadas em bancos de dados

orientados a objetos, bancos de dados relacional, bancos de dados objeto-relacional e em simples arquivos, BROSE (2001).

Será utilizado um simples exemplo de agendamento de reunião para visualizar e entender os conceitos do serviço de persistência PSS, bem como a definição do PSDL para o mesmo.

4.2.1. Definição do PSDL

Nesta seção será explicado como o PSDL define os estados e operações dos objetos armazenados, para combinar a eficiência no acesso aos objetos persistidos e a facilidade no desenvolvimento. Será mostrado como o PSS é utilizado para tornar o estado do objeto persistido em CORBA. Na figura 4.4 foi implementado como exemplo a *Interface Meeting*.

```
Interface Meeting {
  readonly attribute string purpose;
  readonly attribute Participant organizer;
  readonly attribute Room location;
  readonly Date when;

  exception MeetingCanceled {
     string reason;
  }

  participantList getParticipants()
     raises (MeetingCanceled);

  void addParticipnat ( in Participant who)
     raises (MeetingCanceled);

  void removeParticipnat ( in Participant who)
     raises (MeetingCanceled);

  oneway void cancel ( in String reason)
     raises (MeetingCanceled);

  void relocate ( in Room where, in Date  when)
     raises (MeetingCanceled, slotAlready Taken);
  }
```

Figura 4.4- Arquivo PSDL – Interface Meeting

No exemplo foi definida uma classe JAVA separada para representar o estado do objeto MEETING, que poderia ser armazenada em disco. Utilizando a estrutura da linguagem PSDL, serão definidos os tipos de dados que representará esta classe. Para as definições do PSDL, o compilador PSDL estará gerando automaticamente códigos em JAVA para a utilização do PSS na implementação. Para isso, na figura 4.5 está definindo os tipos de dados.

```
Public static Class MeetingState
  Implements java.io.Serializable {

  /*transient data */
  //...
  /* the actual state */
  String purpose;
  String [] participantIORs;
  String organizerIORs;
  String locationIOR;
  Date when;
  boolean canceled = false;
  String cancelReason = null;
   }
```

Figura 4.5- Arquivo PSDL – Interface Class MeetingState

Para a classe *MeetingState* foram definidos quatros atributos e duas variáveis para distribuir o cancelamento das solicitações, são elas *canceled* e *cancelReason* e uma representação da lista dos participantes. Para simplificar, foi utilizado um mesmo tipo de objetos para a classe *Meeting*, BROSE (2001).

```
//file: Meeting.psdl
#include "Office.idl"

module com{
.....

module meetingroom{

  abstract storagetype MeetingState {
     state string purpose;
     readonly state string organizer;;
     state string location;
     state Date when;
     state boolean canceled;
```

```
    state string cancelReason;
    state CORBA::StringSeq participants;
};

  abstract storagehome MeetingState Home of MeetingState {
    factory create (in string purp,
                    in string org,
                    in string loc,
                    in Date when,
                    in Boolean cancld,
                    in string cncl_reason,
                    in CORBA::StringSeq participants );
};
  storagetype MeetingstateImpl
    Implements MeetingState {};
  storagehome MeetingStateHomeImpl of MeetingStateImpl
    Implements MeetingStateHome {};
};
```

Figura 4.6- Arquivo PSDL – Interface Meeting.psdl

Conforme a figura 4.6, a operação *Factory Create* criada na instancia do banco de dados a ser armazenada é uma definição de construção. O compilador PSDL esta habilitado para gerar os códigos requeridos pela linguagem e pelo serviço PSS da implementação, ficando mapeado os tipos de objetos para a interface e classes JAVA. Então o PSDL estará definindo os tipos de objetos e instanciando os mesmos para as classes JAVA na implementação do serviço PSS , BROSE (2001).

4.2.2. Implementação do Exemplo

Para utilizar o serviço de persistência existe a necessidade de se definir alguns métodos para utilização na implementação. Como estes métodos não foram implementados na classe *MeetingState* será então especificadas na classe *MeetingStateImpl,* direcionadas diretamente pela classe *MeetingDefaultServant*, conforme mostrado na figura 4.7.

```
import java.util.*;
import java.io.*;
import org.omg.CORBA;
import org.omg.PortableServer;
import org.omg.PortableServer.POAPackage;
```

```
class MeetingDefaultServant
  extends MeetingPOA{

  private ORB orb;
  private POA poa;
  private Calendar calendar;
  private DateComparator dateComparator;
  private MeetingStateHomeImpl myHome;

  MeetingDefaultServant (ORB orb, POA poá, MeetingStateHomeImpl myHome){
    this.orb = orb;
    this.poa = poa;
    this.myHome = myHome;
    calendar = Calendar.getInstance ();
    datecomparator = new DateComparator ();
};
```

Figura 4.7- Implementação do MeetingServant

O construtor para esta versão da implementação do *MeetingDefaultServant* recebe as definições e argumentos da instancia do banco de dados. Com isso, quando a classe recebe uma requisição o servidor posiciona nos objetos definidos pela instancia no banco de dados. Isto ocorre na utilização do método *myState()* relatada na figura 4.8, que utiliza o identificado PID (*Object ID*) do objeto requisitado e armazenado.

```
  MeetingStateImpl myState(){
    return (MeetingStateImpl) myHome.find_by_short_pi(Object_id () );
)

  public String purpose(){
    return myState().purpose();
)

  public DigitalSecretary organizer(){
    return DigitalSecretaryHelper.narrow(orb.string_to_object (myState().organizer()));
)
```

Figura 4.8- Implementação dos Método do MeetingServant

A implementação do *MeetingDefaultServant* é uma simplificação para o acesso aos dados e objetos, delegando a responsabilidade do gerenciamento e armazenamento para o serviço de persistência PSS. Para finalizar será implementado o *MeetingServer*, que será responsável pelo gerenciamento de sessões com o banco de dados e

armazenamento de objetos, para estar retornando as solicitações do *MeetingDefaultServant*, conforme a figura 4.9.

```
import java.util.*;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CosPersistentState.*;

public class Meeting server {

  public static void main (String[] args) {
    MeetingFactoryImpl meetingFactory;
    Try {
        //Init
        ORB orb = ORB.init (args, null);
        //get Connector Registry
        ConnectorRegistry registry = ConnectorRegistryHelper.narrow
(orb.resolve_initial_registry("PSS"));
        //get Connector
        Connector connector = registry.fin_connector("xx_id");
        //Create session
        Session session = connector.create_basic_session
(org.omg.CosPersistentState.READ_WRITE, "", new Parameter [0]);
        //get root POA
        POA root POA = POAHelper.narrow (orb.Resolve_initial _references
("rootpoa"));
        .....
        //Create a new user
        MeetingDefaultServant mds = new MeetingDefaultServant (orb, meetingPOA,
home);
        .....
        rootPOA.the_POAmANGER().active();
        //Wait for Request
        orb.run() }
}
```

Figura 4.9- Implementação do MeetingServer

4.2.3. Interfaces do PSS

O Persistent State Service (PSS) define uma API para um mecanismo de persistência que pode ser utilizado em diferentes implementações persistentes. Alteração de bancos

de dados para o armazenamento de objetos é uma das conseqüências para a alteração dos códigos PSDL gerados pelas interfaces persistentes, visualizando os diferentes tipos de objetos disponíveis em cada banco de dados. Entretanto estas alterações poderão ser disponibilizadas novamente compilando o código PSDL e liberando a utilização da implementação, tornando assim uma solução totalmente portátil, BROSE (2001).

As interfaces do Persistent State Service (PSS) são definidas no módulo *CosPersistenteState* conforme a figura 4.10 e são declaradas localmente, onde a interface é utilizada na classe MeetingServer conforme a figura 4.9.

```
Module CosPersistentState{

    native StorageObjectBase;
    native StorageObjectFactory;
    native StorageHomeFactory;
    native SessionFactory;
    native SessionPOOLFactory;

    local interface Connector;
      exception NotFound{};

    local interface ConnectorRegistry;
      Connector find_connector (in string implementation_id);
      raises (NotFound);

      void register_connection(in Connector c);

      void unregister_connection(in string implementation_id);
      raises (NotFound);

};
```

Figura 4.10 - Módulo CosPersistentState

Esses dados armazenados precisam da implementação do serviço PSS para serem criados pelos objetos das implementações. Utilizando os tipos de objetos definidos conforme o banco de dados utilizado pela aplicação, gerando o controle e gerenciamento dos objetos pelas classes desenvolvidas.

5. O SERVIÇO DE PERSISTÊNCIA (PSS)

Neste capítulo apresentam-se a especificação e estruturação do serviço de persistência PSS e da implementação desenvolvida para este trabalho, utilizando o estudo de caso do PSS do CORBA.

5.1. Desenvolvimento do Serviço PSS

Nesta etapa apresenta-se o ambiente, especificação e modelos do desenvolvimento da implementação do serviço PSS realizada neste trabalho.

5.1.1. Ambiente de Desenvolvimento

A implementação deste serviço utiliza produtos que estão em conformidade com o padrão CORBA. O projeto foi desenvolvido utilizando a linguagem de programação JAVA, onde foram utilizados no ambiente de desenvolvimento as seguintes ferramentas:

- JDK 1.3 (Java Developer's Kit) Kit de Desenvolvimento Java da Sun;
- TextPad (Editor de Códigos Fontes);
- Visibroker da Borland 4.5.1 (Para NT Server),

Para o projeto, desenvolvimento e teste da aplicação, foi definida uma estrutura física com os seguintes equipamentos :

- Rede local de 10 MB utilizando o protocolo TCP/IP;
- Um servidor Windows NT 4.0 Server (AMD K6 II – 500 MHZ – 256 MB RAM);
- Uma estação Windows 98 (PENTIUM – 166 MHZ – 32 MB RAM).

No servidor está instalado e configurado o JDK 1.3 e o Visibroker 4.5.1, para que seja desenvolvida toda a aplicação. Para os testes e funcionamento do serviço utiliza-se um sistema gerenciador de banco de dados relacional (SGBD). O mesmo serviço será previamente comparado utilizando um sistema gerenciador de banco de dados objeto relacional :

- SQL SERVER 2000 (Banco de Dados Relacional)
- CACHÉ 4 (Banco de Dados Objeto Relacional)

5.1.2. Aspectos para a Implementação do PSS

Para este trabalho foi desenvolvido uma aplicação com o fim de obter um controle de cadastro básico de veículos. Essa aplicação foi desenvolvida em 3 camadas utilizando a arquitetura CORBA, determinando-se assim um ambiente distribuído.

A comunicação entre os objetos neste ambiente distribuído foi operacionalizada através de um *middleware* que implementa a arquitetura CORBA. Esta aplicação é composta por objetos da aplicação e por serviços como o da persistência, para armazenamento dos objetos persistentes da aplicação. Neste caso são utilizados vários objetos da aplicação como o SERVIDOR que será responsável em gerenciar os serviços (CORBA) e requisições dos objetos CLIENTE que contêm toda a interface com o usuário (GUI). No objeto CLIENTE será referenciado o serviço da OPERAÇÃOIMPL que será responsável pela utilização do serviço de persistência desenvolvido (PSS). Têm-se na figura 5.1, toda a implementação da aplicação em Java e o serviço desenvolvido executando um serviço mediador entre o repositório de objetos.

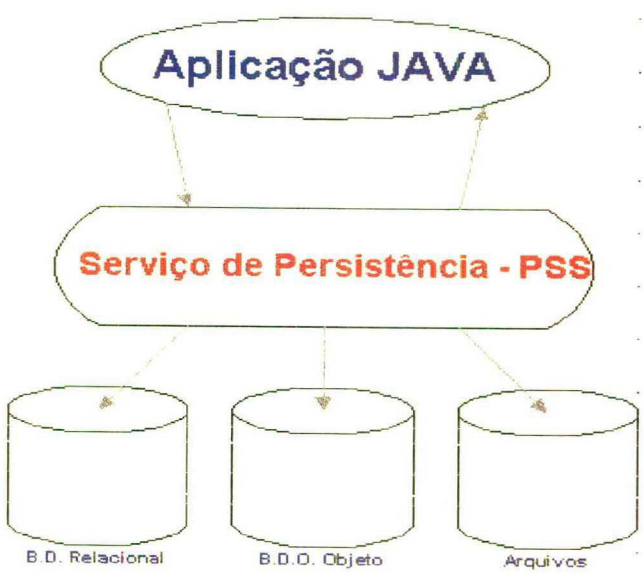

Figura 5.1- Serviço de Persistência PSS

O Serviço de Persistência implementado apresenta-se como o mediador entre a aplicação e o sistema de armazenamento de dados. O *Persistent State Service* (PSS) especificado pelo padrão CORBA, procura facilitar o gerenciamento de mudanças (alteração estrutura de um objeto, novos serviços e novas regras de negócios) evitando a necessidade de re-compilação do serviço.

O serviço acessa dinamicamente a interface da base de conhecimento dos objetos a fim de identificar os seus métodos, atributos e serviços, disponibilizando para a aplicação o acesso a este sistema que pode ser um arquivo simples, em banco de dados relacional ou em banco de dados orientado a objetos.

Os objetos da aplicação solicitam ao serviço de persistência o acesso ao sistema de armazenamento. Os objetos persistentes da aplicação não necessitam conter código de desenvolvimento para as operações de estruturação (criação da classe do objeto), armazenamento (inclusão, alteração e exclusão do objeto) ou recuperação (consulta do objeto).

Na utilização da operação de estruturação, o serviço de persistência (PSS) executará de forma automática a estruturação destas classes dos objetos. Para a operação de armazenamento, o objeto envia uma mensagem ao serviço de persistência (PSS) que será o responsável para gerenciar o PID (Identificador da Persistência) para este objeto. Na operação de recuperação, o objeto apenas envia ao serviço de persistência apenas o PID do objeto solicitado. Este serviço mantém-se ativo no objeto SERVIDOR da aplicação, aguardando as solicitações dos objetos CLIENTE para acessarem o sistema de armazenamento. O serviço de persistência é responsável por gerenciar qualquer solicitação de qualquer estação de rede.

5.1.3. Funcionamento do Serviço PSS Padrão

O serviço de persistência PSS do CORBA surgiu para tornar mais transparente o acesso ao sistema de armazenamento. O PSS fornece uma interface única para armazenar o

estado persistente dos objetos em uma variedade de dispositivos de armazenamento (BDR, BDOO e arquivos). Trata-se de uma interface entre a arquitetura CORBA (*Servants, POAs* entre outras) e as plataformas de armazenamento de dados que permite salvar e restabelecer o estado dos objetos de forma totalmente transparente para os clientes da aplicação.

A nova especificação para o serviço estabeleceu uma linguagem, a PSDL (*Persistent State Definition Language*), que possui construtores os quais definem objetos persistentes e dispositivos de armazenamento. Duas etapas são necessárias para empregar esta linguagem no processo de armazenamento dos objetos persistentes. A primeira consiste em criar, para cada classe persistente da aplicação, um arquivo com extensão PSDL contendo a descrição dos atributos que devem ser armazenados no banco de dados. A segunda caracteriza-se pela compilação e geração dos códigos necessários para armazenar/recuperar os objetos persistentes da aplicação.

Conforme mostrada na figura 5.2, a utilização do sistema gerenciador de banco de dados objeto relacional, neste ambiente proporciona total integração com o serviço de persistência. O sistema de banco de dados CACHÉ 4 realiza a geração dos arquivos PSDL em JAVA, contendo toda a sua estrutura da classe gerada, bem como todos os métodos implementados diretamente no SGBD.

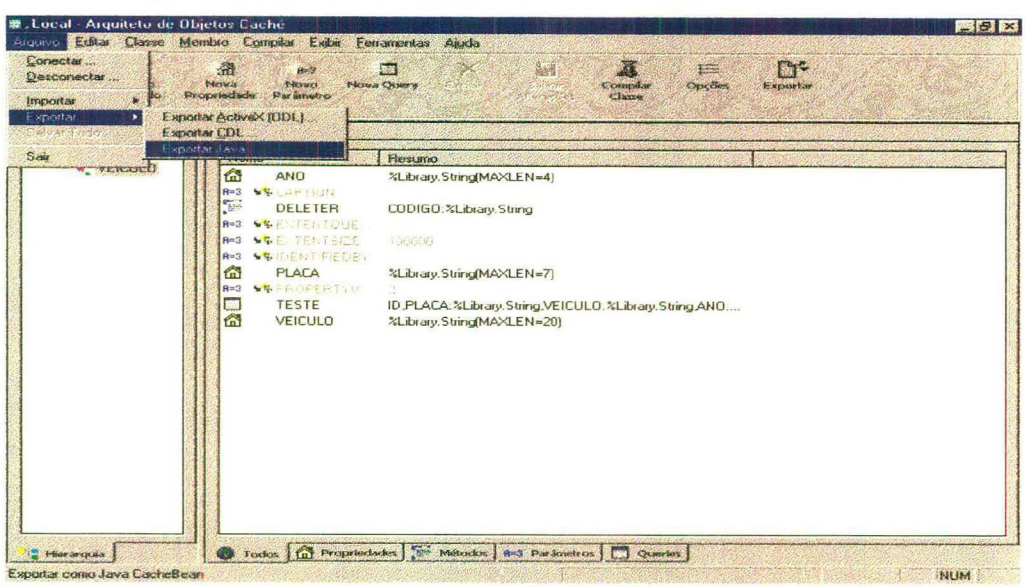

Figura 5.2 Gerar Arquivo PSDL – CACHÉ 4(JAVA)

Assim sendo, para cada classe persistente da aplicação deve existir um arquivo PSDL, mostrado na figura 5.3. Alterações nestas classes, como por exemplo a inclusão de um novo atributo, sem que se altere o arquivo PSDL correspondente, pode comprometer a integridade de armazenamento dos dados.

O trabalho de pesquisa aqui apresentado foi idealizado visando suprimir esta responsabilidade de mapeamento de cada classe para um arquivo PSDL e eliminar a necessidade de geração de código para os objetos persistentes em caso de alteração. Abaixo, um esquema de algoritmo persistente (PSDL) gerado a partir do sistema gerenciador de banco de dados objeto relacional CACHÉ 4 (ANEXO I):

```
import COM.intersys.objects.*;
import COM.intersys.objects.attribute.*;
import COM.intersys.util.SysList;
import COM.intersys.util.CacheException;

public class VEICULO extends Persistent {
/**
Constructor creates a new instance
**/
public VEICULO( ObjectServer os ) throws CacheException {
      super( os, new SysList() );
   }
.....
/**
    * Java implementation of DELETER
    *
   **/
   public void DELETER(String CODIGO) throws CacheException {
      SysList    _args;
      synchronized (m_objectServer) {
         _checkObject( false );
         _args = new SysList();
         _args.set( CODIGO, 0 );
         m_objectServer.invokeClassMethod( "User.VEICULO", "DELETER", _args,
false );
      }
   }
```

Figura 5.3 Arquivo PSDL - Veículo

5.2. Funcionamento do Serviço Implementado

Como o desenvolvimento deste serviço de persistência especificado pelo padrão CORBA, utilizando a concepção do PSS, abrangerá todos os sistemas gerenciadores de banco de dados, não se pode estar utilizando o processo descrito na figura 5.1. Pois somente seria possível em um sistema gerenciador de banco de dados objeto relacional ou orientado a objetos, que já possuem estas características de geração do arquivo PSDL utilizado no PSS. Por isso, se faz necessário adotar um procedimento para que o serviço possa ser utilizado em diversos sistemas gerenciadores de banco de dados e até em arquivos relacionais, BUZATO (1998).

Dessa forma, o Serviço de Persistência (PSS) implementado apresenta uma base de dados denominado base de conhecimento, que contém a classe base de estrutura na qual deverá ser previamente configurada pelo Administrador de Banco de Dados, com a respectiva estrutura (Conforme Anexo II):

- **Base de Estrutura:** Esta base de estrutura contém toda a relação das classes persistentes da aplicação especificando todos os atributos e qual o seu tipo, que podem estar sendo utilizados pelo objeto na aplicação. Esta base é toda a estrutura do serviço de persistência. Com esta estruturação o serviço de persistência não estará dependendo do sistema gerenciador de banco de dados para gerar o arquivo PSDL, integrando assim vários bancos de dados.

No desenvolvimento de cada classe persistente da aplicação deveria existir um arquivo de estrutura em PSDL, conforme concepções do *Persistent State Service*, sendo utilizada somente a base de estrutura pelo serviço de persistência desenvolvido, BUZATO (1998).

Com o aproveitamento deste recurso de persistência e da base de conhecimento, conforme mostrado na figura 5.4, o serviço de persistência (PSS) possui a capacidade de acessar os dados de um objeto em tempo de execução na sua referida base de conhecimento (base estrutura) e estruturar os serviços em conformidade com os atributos e seus respectivos tipos e, finalmente, armazená-los em um sistema gerenciador de banco de dados.

O serviço, descrito na figura 5.4, acessa em tempo de execução a interface dos objetos, desta maneira obtém-se, dinamicamente, conhecimento de como utilizar os métodos e atributos dos objetos.

Figura 5.4- Definição da Base de Conhecimento

O funcionamento do serviço de persistência (PSS) desenvolvido para armazenamento do estado de um objeto pode ser descrito, de forma sucinta, do seguinte modo:

- O serviço (PSS) recebe uma mensagem de solicitação de um objeto da aplicação para acessar ou executar qualquer operação utilizando o banco de dados (processo 1, da figura 5.5);

- Através da solicitação do objeto da aplicação, o serviço (PSS) irá gerar um processamento interno e acessar a base de armazenamento, conforme o serviço solicitado irá gerar um PID (Identificador da Persistência) para sua execução (processo 2);

- Conforme o tipo de operação solicitado pelo objeto da aplicação o serviço de persistência (PSS) irá acessar a base de conhecimento para executar de forma dinâmica a sua instrução (processo 3);

- Em seguida, recupera-se da base de conhecimento a instrução gerada e executa na referida base de dados, armazenando ou consultando um objeto solicitado pela aplicação (processo 4);

- Assim, o serviço de persistência é capaz de acessar o objeto na memória, recuperar o estado atual do objeto e retornar para o objeto da aplicação a sua solicitação (processo 5).

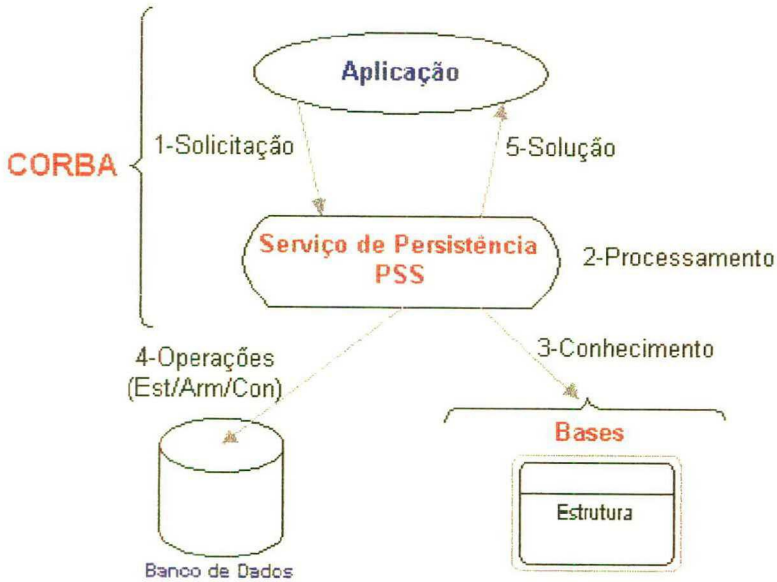

Figura 5.5- Funcionamento do Serviço Persistência PSS

No entanto, tem-se consciência de que, o acesso dinâmico à esta interface da base de conhecimento do objeto demanda um certo tempo, o que pode tornar o serviço implementado mais lento quando comparados aos serviços de persistência que acessam objetos pré-definidos no processo de compilação. Não é objetivo deste trabalho analisar este desempenho através de comparações entre o tempo de acesso estático e o acesso dinâmico a essas estruturas implementadas.

5.3. Fases da Implementação do Serviço

Para este trabalho foi desenvolvida uma aplicação que é composta por objetos em uma estrutura em N camadas utilizando a arquitetura CORBA, determinando uma aplicação em ambiente distribuído, BUZATO (1998). Conforme os tópicos da tabela 5-1, pode-se ver cada camada da aplicação especificada.

CÓDIGO FONTE	*DESCRIÇÃO*
Server.Java	Servidor da Aplicação, desenvolvido utilizando toda a arquitetura Corba, onde receberá a conexão do Cliente
Cliente.Java	Cliente, gerenciamento da conexão com servidor Corba
Frm_Veiculo.Java	Formulário Veículo, interface com o usuário utilizando o serviço de persistência PSS
Operação_IMPL.Java	Todas as operações relacionadas na aplicação com o gerenciamento da arquitetura CORBA
PSDL.Java	Classe onde teremos todas as conexões com o Banco de Dados para utilizar os recursos do serviço de persistência PSS
Persistente.Java	Classe onde está implementado o serviço de persistência PSS utilizando os métodos da Classe Veículo e a Base de Conhecimento

TABELA 5.1- OBJETOS DA APLICAÇÃO DESENVOLVIDOS

Conforme os objetos da aplicação desenvolvidos, mostrar-se-á na figura 5.6 como o trabalho foi desenvolvido. A figura 5.6 mostra a interação entre os objetos e os níveis definidos para que ocorra o serviço de persistência de objetos. Finalizando o processo no repositório de objetos, banco de dados.

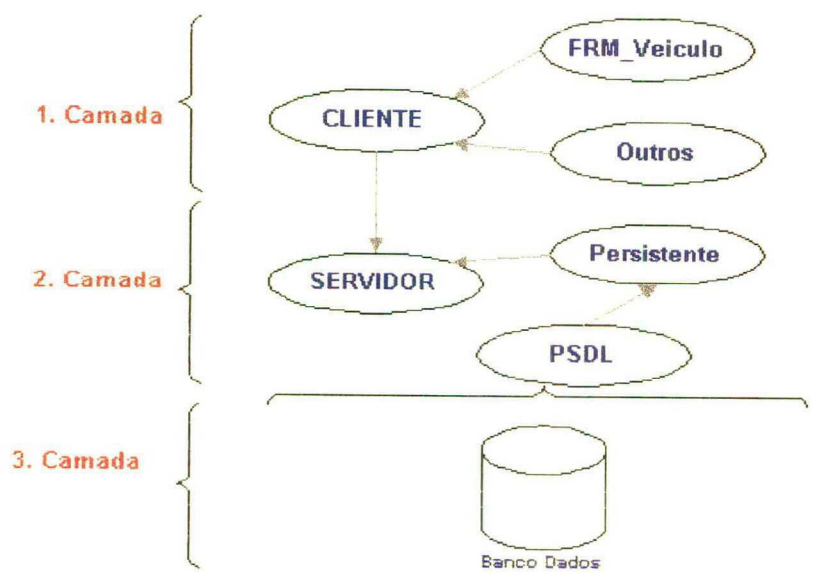

Figura 5.6- Fases e Processos da Implementação

5.3.1. Servidor da Aplicação (SERVER)

No objeto SERVIDOR, mostrado na figura 5.7 foi desenvolvida toda a base para a operação em N camadas. Verifica-se também toda a estrutura para a conexão de objetos utilizando a arquitetura CORBA, onde o servidor controla e gerencia todas as conexões dos Clientes no sistema.

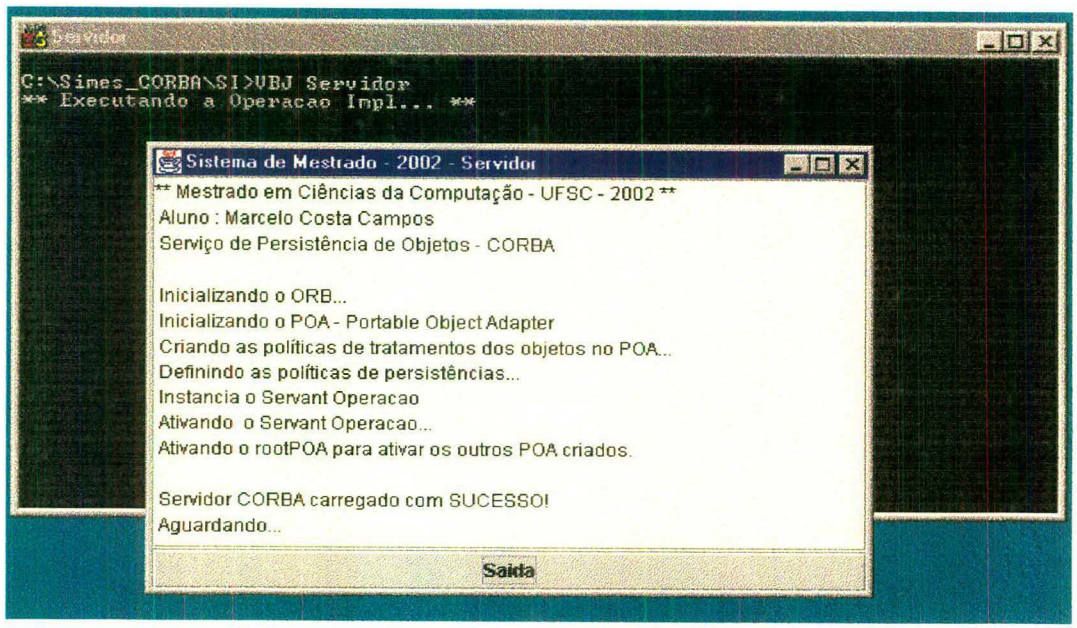

Figura 5.7- Servidor da Aplicação

Conforme mostrado na figura 5.7, o objeto servidor estará gerenciando a conexão do objeto cliente que é realizada através da configuração do CORBA. Também instanciando o objeto que controla todos os métodos e processos que são realizados neste desenvolvimento. A seguir na figura 5.8, a rotina do objeto servidor que realiza a conexão com o CORBA e com o objeto responsável pelo gerenciamento dos métodos do sistema.

```
import java.awt.*;
import java.io.*;
import java.awt.event.*;
import javax.swing.*;
import org.omg.PortableServer.*;
```

```
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValueHelper;

public class Servidor extends JFrame {

.....

//*********************************************************************
 //Metodo Inicializador da Classe
 public static void main (String args[])
 {

    Servidor app = new Servidor();
    app.addWindowListener
    (
      new WindowAdapter()
      {
        public void windowClosing( WindowEvent e )
        {
          System.exit( 0 );
        }
      }
    );

    //app.runServer(args[]);

  try
  {
          //Inicializa o ORB
          org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
//      display.append( "\n Inicializando o ORB..." );
   //Inicializa o POA
      POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
//      display.append( "\n Inicializando o POA - Portable Object Adapter" );
      // Criando as políticas de tratamentos dos objetos no POA - Persistentes
      org.omg.CORBA.Any any = orb.create_any();
      BindSupportPolicyValueHelper.insert(any, BindSupportPolicyValue.BY_INSTANCE);
      org.omg.CORBA.Policy                                    bsPolicy                                =
orb.create_policy(com.inprise.vbroker.PortableServerExt.BIND_SUPPORT_POLICY_TYPE.value,
any);
//      display.append( "\n Criando as políticas de tratamentos dos objetos no POA..." );
      org.omg.CORBA.Policy[] policies =
      {
       rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT), bsPolicy
      };

//      display.append( "\n Definindo as políticas de persistências..." );
      // Create myPOA with the right policies
      POA operacaoPOA = rootPOA.create_POA("objOperacao", rootPOA.the_POAManager(),
```

```
                                policies );
        // Criando o Servant
        OperacaoImpl operacaoServant = new OperacaoImpl();
//        display.append( "\n Instancia o Servant Operacao" );
        // Decide on the ID for the servant
        byte[] operacaoId = "poaOperacao".getBytes();
        // Activate the servant with the ID on myPOA
        operacaoPOA.activate_object_with_id(operacaoId ,operacaoServant);
//        display.append( "\n Ativando  o Servant Operacao..." );
    //Msg do Servidor ready
        // Activate the POA manager
        rootPOA.the_POAManager().activate();
//        display.append( "\n Ativando o rootPOA para ativar os outros POA criados." );
    //Aguarda chegada dos servicos
//        display.append( "\n Servidor carregado com SUCESSO!" );
//        display.append( "\n Aguardando..." );
        orb.run();

    }
.............
```

Figura 5.8- Objeto Servidor Desenvolvido

5.3.2. Cliente da Aplicação (CLIENTE)

No objeto CLIENTE desenvolveu-se toda a interface com o usuário e estabelecendo assim uma base para a operação em camadas. Este cliente da aplicação foi implementado para a conexão de objetos utilizando a arquitetura CORBA, onde o servidor controla e gerencia toda a conexão do objeto cliente e este, depende do servidor para poder acessar as rotinas e métodos definidos para este sistema.

O objeto cliente proporciona o controle sobre os outros objetos desenvolvidos neste sistema, como o objeto FRM_VEICULO que é o responsável, pela execução dos métodos de inclusão e exclusão do sistema. Conforme figura 5.9.

Verifica-se que o objeto FRM_CONSULTA é responsável pela operação de consulta destas informações inseridas no banco de dados. Através deste formulário, o sistema interage (conforme mostrado na figura 5.6) com o sistema de armazenamento de objetos.

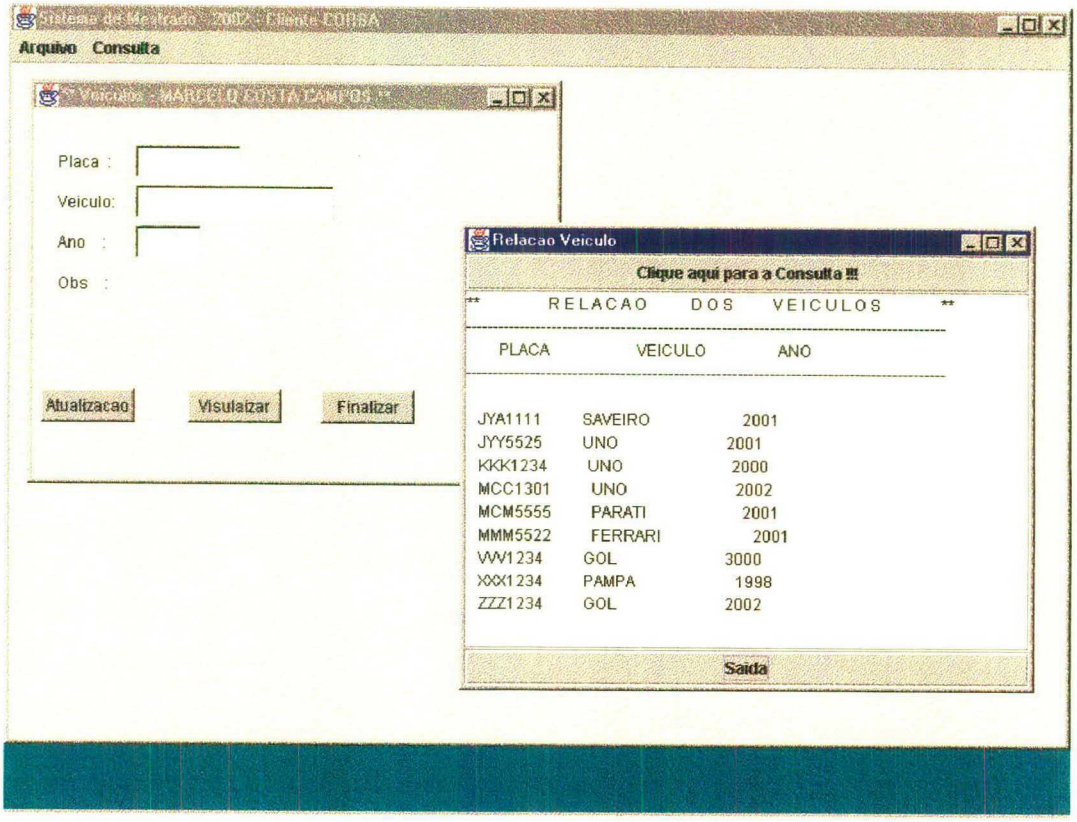

Figura 5.9- Cliente da Aplicação

5.3.3. Métodos da Implementação

Conforme especificação do CORBA, na figura 5.10 observa-se o desenvolvimento do arquivo VEICULO.IDL que gera todos o métodos utilizados no desenvolvimento da implementação, gerenciado pelo serviço *OSAGENT* do *VisiBroker*.

Com estes métodos implementados o sistema realiza todos os processos necessários para este trabalho. Na figura 5.10 o modelo do método da implementação, KOOSIS (1999).

```
//MÓDULO PARA CONFIGURAÇÃO DO IDL
module Veiculo {
  interface Operacao
```

```
  {
    string Incluir  (in string a, in string b, in string c);
    string Pesquisar(in string a);
    string Excluir(in string a);

  };

};
```

Figura 5.10- Configuração do Módulo IDL

Conforme especificado na figura 5.10, tem-se definido os métodos de Inclusão (Incluir dados no sistema armazenador) e Exclusão (Excluir dados no sistema armazenador), bem como a rotina de pesquisa que foi utilizado na implementação.

5.3.4. Implementando o Serviço de Persistência PSS

O objeto PERSISTENTE desenvolvido é o responsável pelo gerenciamento e controle do serviço de persistência PSS para esta aplicação. Através deste objeto que contém toda a estrutura que contempla a utilização da base de conhecimento (base de estrutura), o serviço tende a centralizar o controle de acesso à base de dados objeto relacional e base de dados relacional, conforme o método utilizado pela aplicação, TAMASIA (2002).

Então todas as operações realizadas nesta aplicação tem como parte do seu processo a utilização deste objeto para a correta execução das rotinas do sistema.

Observa-se na figura 5.11, a implementação do objeto persistente. Para completar a estrutura para à utilização do serviço de persistência foi desenvolvido o objeto PSDL, que faz relacionamento direto com o objeto descrito acima, fechando assim todo o ciclo de utilização desta aplicação. KOOSIS (1999).

```
Import java.lang.reflect.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
```

```
import javax.swing.border.*;
import java.sql.*;

public class Persistente extends JFrame
{
   //protected objectID;
   private   Connection conn;

   public Persistente()
   {
      try
      {
       //Preparando as Variaveis para a Conexao com o Banco de Dados
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       conn = DriverManager.getConnection("jdbc:odbc:VEICULOJDBC","sa","");
                }
      catch (Exception e)
      {
         System.out.println("Erro na Conexao JDBC");
         e.printStackTrace();
                }
   }
   //Metodo Insercao de Dados da Classe
   public void insert ()
   {
                //long objectID     = this.obtemProximoIdentificador();
                //String nomeClasse = this.getClasse().getName();
                String nomeClasse = "VEICULO";
.....
   //1 - Recebe qual a Classe que ira implementar
   //2 - Retorna a string Query com a consulta SQL
   public String select( String classe)
   {
     try
     {
       //Consulta SQL para Levantamento dos Campos para a selecao
       //na tabela ESTRUTURA
       Statement stmt   = conn.createStatement();
       String      sql      = "SELECT CAMPO_EST FROM ESTRUTURA WHERE
ID_EST = '"+ classe +"' AND USADO_EST = 'S' ORDER BY SEQ_EST";
       String    campos = " ";
       //Recebbe o resultado da Pesquisa SQL
       ResultSet  rs  = stmt.executeQuery(sql);
       sql = " ";
       boolean more   = rs.next();

       // Fica no While ate o final dos Campos
                  do {
                    campos += rs.getString(1) + ",";
```

```
            } while ( rs.next() );
    //Incrementando as Váriaveis
    if(campos.length() > 0)
    {
                campos = campos.substring(0,campos.length() - 1);
    }
    //Agora Monta o Comando SQL para repassar pelo Return
    sql = "SELECT "+ campos +" FROM VEICULO ";
    return sql;
  }
 catch (Exception e)
  {
    return "ERRO";
 }
```

Figura 5.11- Objeto Persistente Desenvolvido

5.3.5. Serviço do VisiBroker

Para o desenvolvimento desta aplicação e obedecendo a especificação do CORBA, foi utilizado o serviço *OSAGENT* do *Visibroker* conforme apresentado na figura 5.12. Este serviço é o responsável pela comunicação e troca de serviços entre o objeto servidor e o objeto cliente do sistema.

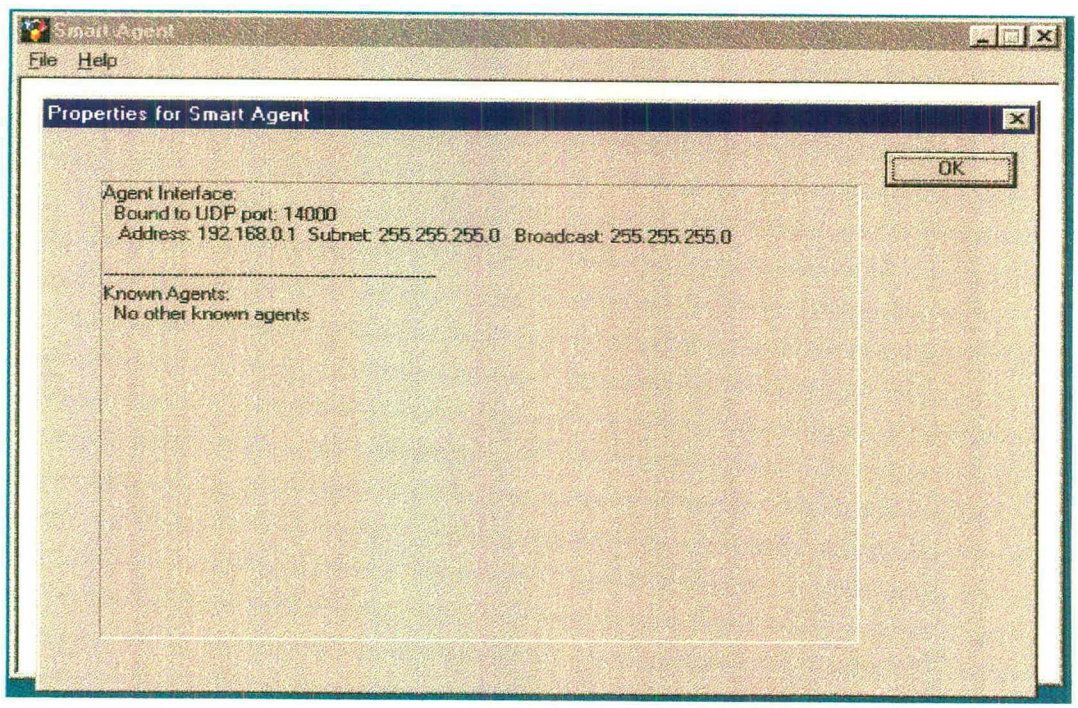

Figura 5.12- Serviço OSAGENT do VisiBroker

Conforme utilização do sistema, gerando troca de informações entre os objetos cliente e servidor, o serviço OSAGENT do Visibroker gera várias informações gerenciais que são armazenadas no LOG, conforme mostrado na figura 5.13.

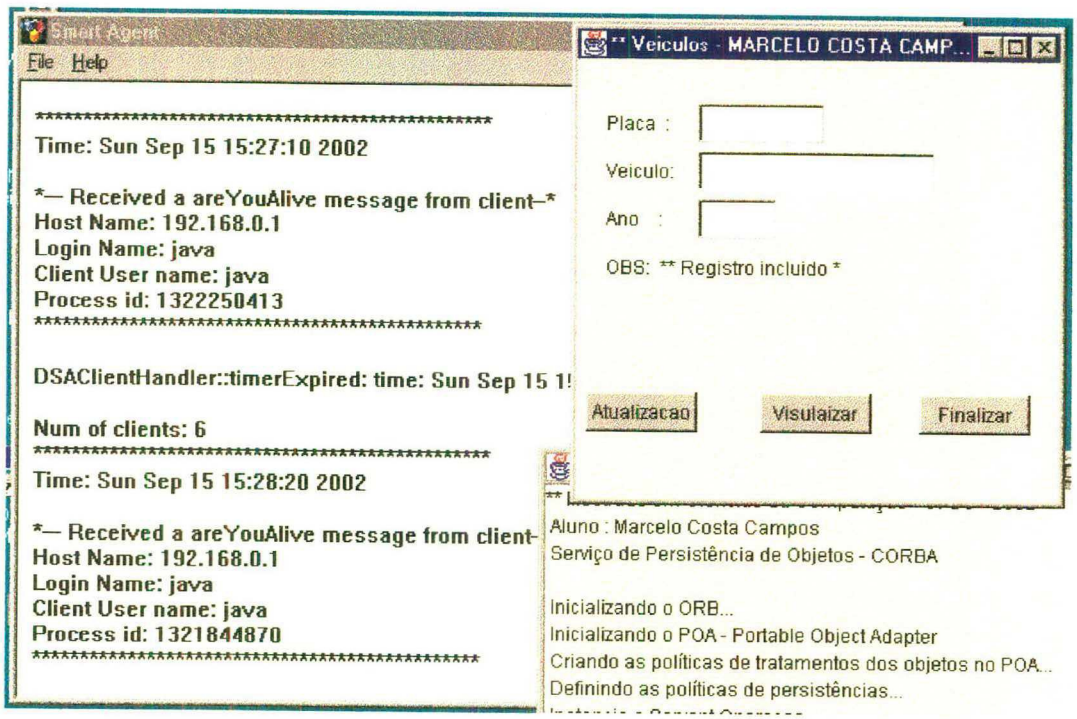

Figura 5.13- LOG de eventos do serviço OSAGENT do VisiBroker

5.4. Importância do Serviço de Persistência (PSS)

Para o processo de desenvolvimento de sistemas tende-se que a programação orientada a objetos se tornará umas das principais tecnologias que orientará os princípios de desenvolvimento de software. A programação orientada a objetos é caracterizada pelo suporte a herança e polimorfismo, permitindo que novos objetos sejam criados como especializações de objetos já existentes, SILVA (2002). Esses objetos já possuem interfaces bem definidas e encapsuladas, que permitem serem tratadas como componentes de software independentes, onde podem servir como blocos para construir aplicações mais amplas.

Seria interessante nunca desenvolver objetos, quando da necessidade de um novo objeto para a aplicação deveria realizar uma pesquisa em um catálogo de objetos para a

aquisição do mesmo para utilizá-lo como bloco de construção, SANTOS (1999). Se o objeto não estiver de acordo com a necessidade da aplicação, utilizaria a herança e polimorfismo para desenvolver um novo objeto. Mas estes componentes terão que se interagir com um grande número de tecnologias, para resolver problemas genéricos de negócios.

Uma das respostas a este dilema é provida por conjuntos de serviços de objetos e arquitetura de objetos, que possam ser especialmente desenvolvidos para ambientes genéricos e desenvolver componentes onde estes serviços estejam disponíveis. O componente então não precisaria possuir códigos específicos para a sua comunicação em rede, ele iria assumir automaticamente que a própria arquitetura seria encarregada de assumir este serviço, assim como não precisaria mais possuir comandos para acesso em banco de dados, pois será gerenciado pelo serviço de persistência em objetos, SILVA (2002).

Portanto, para solucionar o re-trabalho precisa-se de padrões para a arquitetura de objeto e para as suas interfaces dos serviços, onde esse é o trabalho do OMG (*Object Management Group*). Gerenciando os serviços para a sua padronização e que o serviço de persistência de objetos será um fator crítico de sucesso para aplicações dos componentes de softwares. Conforme este serviço de persistência se torna mais integrado com outros serviços como de consulta, segurança, transações entre outros, teremos uma tecnologia poderosa para o desenvolvimento de componentes de softwares reutilizáveis, SILVA (2002).

O padrão CORBA especifica um serviço de persistência para cada classe da aplicação, que contenha objetos persistentes, para tanto deve existir um arquivo PSDL. Com o serviço de persistência CORBA é necessário um controle rígido da configuração de versões das classes do sistema, a fim de garantir que mudanças nos atributos de classes, que contenham objetos persistentes, sejam também refletidas nos respectivos arquivos PSDL.

Considera-se persistência como algo extremamente necessário em linguagens de programação orientadas a objetos, bem como em outros tipos de linguagens tradicionais.

Analisando-se algumas linguagens, percebe-se que a implementação da persistência se dá em três abordagens: arquivos, RDBMS e ODBMS. O serviço de persistência tende a gerar:

- Uma demanda nas características de interoperabilidade, reusabilidade e portabilidade nas aplicações;

- Produtos para persistência devem aumentar a produtividade e reduzir os custos de manutenção mantendo as premissas da Tecnologia de Objetos;

- Prover persistência arquitetural é um trabalho complexo;

- Nenhuma tecnologia é uma ilha: utilizar arquiteturas abertas;

- Alguma arquitetura de persistência é melhor do que nenhuma.

6. CONCLUSÃO

Conclui-se este trabalho de pesquisa, apresentando os resultados obtidos a contribuição ao meio cientifico, bem como abertura para trabalhos futuros. Onde este trabalho apresentou o desenvolvimento de aplicações distribuídas utilizando o serviço de persistência PSS na arquitetura CORBA.

Ficou patente que atualmente, as equipes de desenvolvimento de software das empresas concentram seus esforços na análise, projeto e implementação dos objetos de negócios. Estes são objetos específicos de uma aplicação para atender aos requisitos dos usuários. No caso de uma aplicação para o desenvolvimento do serviço de persistência PSS, este pode ser considerado um objeto de negócio.

Com este trabalho, uma das contribuições aos meios cientifico e acadêmico foi que o serviço PSS desenvolvido poderá reduzir o trabalho do desenvolvedor evitando que ele tenha que mapear cada classe para um arquivo PSDL e permitindo que ele concentre seus esforços na especificação dos objetos de negócio. Os objetos persistentes da aplicação não precisam conter código para operações de armazenamento ou recuperação. Cada objeto de negócio possui métodos de acesso ao sistema de armazenamento de dados, mas o código interno desses métodos contém simplesmente uma solicitação de serviço ao serviço de persistência PSS desenvolvido.

Ressalta-se neste estudo um aspecto importante deste serviço desenvolvido, que é a utilização da base de conhecimento. Pois ao explorar o conteúdo desta base de conhecimento (base de estrutura) verificou-se que todas as alterações em atributos ou métodos de um objeto são imediatamente refletidas em suas interfaces e, em conseqüência, são automaticamente visíveis para o serviço de persistência. Por isso, uma re-compilação do serviço de persistência não se faz necessária.

Este trabalho relatou ainda sobre o projeto para desenvolvimento de um serviço de persistência (PSS), originado da arquitetura CORBA e tornou como base todas as suas características. O projeto valeu-se de algumas percepções da especificação PSS, como a

transparência, quanto ao tipo de sistema de armazenamento utilizado, redução da carga de trabalho dos desenvolvedores, auxílio no gerenciamento de mudanças e geração de uma demanda nas características de interoperabilidade, reusabilidade e portabilidade nas aplicações.

Ficou evidente também que neste estudo, a heterogeneidade existente de equipamentos, de sistemas operacionais e de autoridade obtém o surgimento de especificações abertas, devido à necessidade de padronização desses sistemas. Essa mesma heterogeneidade gera á necessidade de fornecer às aplicações uma série de mecanismos e serviços, como o de persistência (PSS), para transparências de distribuição.

Outra contribuição será a abertura de novos temas para trabalhos futuros, onde se propõe a integração do serviço de persistência com os outros serviços e facilidades do CORBA, pois esta integração do CORBA poderá ser explorada para desenvolvimento de diversas aplicações. Principalmente as aplicações que possuem os conceitos de ERP e CRM. Disponibilizando assim estes recursos para diferentes plataformas e banco de dados, integrando os serviços a novas tecnologias para disponibilizar ao mercado.

Enfim, nesse contexto, o presente trabalho conseguiu atingir seus objetivos ao contribuir com a aplicabilidade de conceitos pertinentes à área de Sistemas Distribuídos. Baseando-se em métodos, ferramentas e procedimentos no processo de desenvolvimento do serviço de persistência PSS em um ambiente distribuído com a arquitetura CORBA. Daí-se a consideração deste tema como uma alternativa para o armazenamento dos objetos em ambientes distribuídos, com o fim de melhorar as vantagens na utilização de aplicações distribuídas.

REFERÊNCIAS BIBLIOGRÁFICAS

ALCALDE, Eduardo. Informática Básica. 1. ed. São Paulo: Makron Books, 1991.

BUZATO, Luiz; RUBIRA, Carlos. Construção de Sistemas Orientados a Objetos. 1. ed. São Paulo: Editora 11, 1998.

BROOKSHER, J. Glenn. Ciência da Computação: Uma Visão Abrangente. 5. ed. trad. Cheng Mei Lee. Porto Alegre: Editora Bookman, 2000.

BROSE, Gerald; VOGEL, Andreas. Java Programming with Corba. 2. ed. New York: Editora John .Wiley & Sons, 2001.

COSTA, Marcos Mota. Desenvolvimento de aplicações com Java e CORBA. Centro-Oeste: Anais da III Escola Regional de Informática do Centro-Oeste, 2000.

FERREIRA, M. G. V. Uma arquitetura flexível e dinâmica para objetos distribuídos aplicada ao Software de Controle de Satélites. Tese de Doutorado. Computação Aplicada. INPE. São José dos Campos, 2001.

KHOSHAFIAN, Setrag. Banco de Dados Orientado a Objetos. 2. ed. New York: Editora John Wyley & Sons, 1994.

KOOSIS, Donald; KOOSIS, David. Programação com Java. 1. ed. Rio de Janeiro: Editora Campus, 1999.

LINS, José Geraldo. Desenvolvimento de Sistemas Persistentes em Java com JDBC. Artigo Científico, UFP. Paraná: UFP, 2000.

MCCARTY, Bill; DORION, Luke Cassady. Java Distributed Objects. 1. ed. New York: Editora Sams, 1999.

NASSU, Eugenio. Bancos de Dados Orientados a Objetos. 1. ed. Porto Alegre: Editora Edgar Blucher, 1999.

ORFALI, Robert; HARKEY, Dan. Instant Corba. 2. ed. New York: Editora John Wiley & Sons, 1997.

OTTE, Randy. Understanding CORBA : The Common Object Request Broker Architecture. 1. ed. New Jersey: Editora Prentice Hall PTR, 1996.

PEREIRA, Patrícia Maria. Serviço de Persistência para Ambientes Distribuídos Explorando os Recursos de Repositório de Interfaces. Artigo Científico, Instituto Nacional de Pesquisas Espaciais - INPE. São José dos Campos. 2000.

REESE, George. JDBC e Java: Programação para Banco de Dados. 1. ed. New York: Editora Berkeley, 2001.

SANTOS, Luiz Gaspar dos. Objetos Distribuídos. 1° Edição. Paraná: UFPR ,1999.

SESSIONS, Roger. Object Persistence. 1. ed. New Jersey: Prentice Hall Inc, 1996.

SILVA, Aridio. Dominando a Tecnologia de Objetos. 1. ed. São Paulo: Book Express, 2002.

TAMASIA, Roberto; GOODRICH, Michael T. Estrutura de Dados e Algoritmo em Java. 2. ed. Porto Alegre: Editora Bookman, 2002.

TEIXEIRA, Jose Helvecio; MOURA, Jose Antão Beltrão. Do Mainframe para a Computação Distribuída: Simplificando a Transição. 1. ed. São Paulo: Editora IBPI PRESS, 1996.

THOMPSON, Marco Aurélio. Java2 e Banco de Dados. 1.ed. São Paulo: Editora Érica, 2002.

ANEXO A - CÓDIGO PERSISTENTE (JAVA)

```
/ * Cache' Java Class Generated 10:52PM  04 Aug 2002 for class User.VEICULO
 **/
import java.util.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.util.Hashtable;
import java.math.BigInteger;
import java.math.BigDecimal;
import COM.intersys.objects.*;
import COM.intersys.objects.attribute.*;
import COM.intersys.util.SysList;
import COM.intersys.util.CacheException;

public class VEICULO extends Persistent {
/**
Constructor creates a new instance
**/
public VEICULO( ObjectServer os ) throws CacheException {
      super( os, new SysList() );
   }
.....
* _close closes an Oref
   public void _close() throws CacheException {
      synchronized (m_objectServer)  {
         super._close();
      }
   }

   /**
    * Property Storage
   **/
   private StringAttr m_ANO;
   private StringAttr m_PLACA;
   private StringAttr m_VEICULO;

.....
   /**
    * Initialize the class
    *
   **/
   protected String _initClass( Hashtable h ) {
      h.put( "ANO", (m_ANO = new StringAttr( this, "ANO" )) );
      h.put( "PLACA", (m_PLACA = new StringAttr( this, "PLACA" )) );
      h.put( "VEICULO", (m_VEICULO = new StringAttr( this, "VEICULO" )) );
      super._initClass( h );
```

```
        return "User.VEICULO";
    }
 ........
    /**
     * Java implementation of DELETER
     *
    **/
    public void DELETER(String CODIGO) throws CacheException {
        SysList    _args;

 ..........

        synchronized (m_objectServer) {
            _checkObject( false );
            _args = new SysList();
            _args.set( CODIGO, 0 );
            m_objectServer.invokeClassMethod(  "User.VEICULO",  "DELETER",  _args,
false );
        }
    }

    /**
     * Query initiator for query TESTE
     *
    **/
    public ResultSet Query_TESTE(String COD) throws CacheException {
        ResultSet _rs;

        _rs = new ResultSet( m_objectServer, "User.VEICULO", "TESTE" );
        _rs.setString( 1, COD );

        return _rs;
    }

/**
 * End-of-file
**/
```

ANEXO B – BASE DE CONHECIMENTO

- CLASSE ESTRUTURA

ATRIBUTO	TIPO	FINALIDADE
Id	Char 03	Identificador do Objeto
SEQ_EST	Int	Seqüência da Estrutura
CAMPO_EST	Char 10	Descrição do Atributo da Classe
TIPO_EST	Char 1	Tipo do Atributo da Classe
TAMANHO_EST	Int	Definição do tamanho do Atributo da Classe
DECIMAL_EST	Int	Definição do tamanho Decimal do Atributo da Classe
USADO_EST	Char 1	Definição de Atributo utilizado na Classe (S/N)
NIVEL_EST	Char 1	Definição do nível do Atributo

TABELA B.1- CLASSE DE ESTRUTURA

ANEXO C – FONTES DO SISTEMA

1. FONTE SERVER.JAVA

```
import java.awt.*;
import java.io.*;
import java.awt.event.*;
import javax.swing.*;
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValueHelper;
public class Servidor extends JFrame {
  //Variaves de Trabalho da Tela
  private JButton b1;
  private JTextArea display;

  public Servidor()  {
    super( "Servidor" );
    //Adicionando o Botão
    b1 = new JButton( "Saida" );
    b1.addActionListener(new java.awt.event.ActionListener()
    { public void actionPerformed(ActionEvent e)
      { b1_actionPerformed(e); }
    } );
    getContentPane().add( b1, BorderLayout.SOUTH );
    display = new JTextArea();
    getContentPane().add( new JScrollPane( display),BorderLayout.CENTER);
    setTitle("Sistema de Mestrado - 2002 - Servidor");
    setSize( 450, 300 );
    display.setBackground(SystemColor.info);
    display.append( "** Mestrado em Ciências da Computação - UFSC - 2002 **");
    display.append( "\n Aluno : Marcelo Costa Campos");
    display.append( "\n Serviço de Persistência de Objetos - CORBA ");
    display.append( "\n ");
    display.append( "\n Inicializando o ORB..." );
    display.append( "\n Inicializando o POA - Portable Object Adapter" );
    display.append( "\n Criando as políticas de tratamentos dos objetos no POA..." );
    display.append( "\n Definindo as políticas de persistências..." );
    display.append( "\n Instancia o Servant Operacao" );
    display.append( "\n Ativando  o Servant Operacao..." );
    display.append( "\n Ativando o rootPOA para ativar os outros POA criados." );
    display.append( "\n " );
    display.append( "\n Servidor CORBA carregado com SUCESSO!" );
    display.append( "\n Aguardando..." );
    show();
  }
//*****************************************************************
  //Metodo de Evento do Botao
```

```
  void b1_actionPerformed(ActionEvent e) {
     // Mensagem de Desconecao
     display.append( "\n-------------------------------------");
     display.append( "\n AGUARDE - DESCONECTANDO O SERVIDOR... ");
     int result;
     result = JOptionPane.showConfirmDialog(this,"Deseja realmente Desconectar o
Servidor ? ");
     if (result == JOptionPane.YES_OPTION)
     {   System.exit( 0 );   }
  }
//*********************************************************************
  //AQUI COMECA O PROGRAMA NOVO
//*********************************************************************
  //Metodo Inicializador da Classe
  public static void main (String args[]) {
     Servidor app = new Servidor();
     app.addWindowListener    (
       new WindowAdapter()     {
         public void windowClosing( WindowEvent e )          {
           System.exit( 0 );
         }
       }
     );
     //app.runServer(args[]);
  try
   {
       //Inicializa o ORB
       org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
//Display.append( "\n Inicializando o ORB..." );
       //Inicializa o POA
      POA                                rootPOA                                =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
//       display.append( "\n Inicializando o POA - Portable Object Adapter" );
      // Criando as políticas de tratamentos dos objetos no POA - Persistentes
      org.omg.CORBA.Any any = orb.create_any();
 BindSupportPolicyValueHelper.insert(any,
BindSupportPolicyValue.BY_INSTANCE);
      org.omg.CORBA.Policy                          bsPolicy                    =
orb.create_policy(com.inprise.vbroker.PortableServerExt.BIND_SUPPORT_POLIC
Y_TYPE.value, any);
//          display.append( "\n Criando as políticas de tratamentos dos objetos no
POA..." );
      org.omg.CORBA.Policy[] policies =
      {
       rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
bsPolicy
      };
//       display.append( "\n Definindo as políticas de persistências..." );
     // Create myPOA with the right policies
```

```
    POA operacaoPOA = rootPOA.create_POA("objOperacao",
    rootPOA.the_POAManager(), policies );
          // Criando o Servant
          OperacaoImpl operacaoServant = new OperacaoImpl();
    //      display.append( "\n Instancia o Servant Operacao" );
          // Decide on the ID for the servant
          byte[] operacaoId = "poaOperacao".getBytes();
          // Activate the servant with the ID on myPOA
          operacaoPOA.activate_object_with_id(operacaoId ,operacaoServant);
    //      display.append( "\n Ativando  o Servant Operacao..." );
           //Msg do Servidor ready
          // Activate the POA manager
          rootPOA.the_POAManager().activate();
    //   display.append( "\n Ativando o rootPOA para ativar os outros POA criados." );
           //Aguarda chegada dos servicos
    //      display.append( "\n Servidor carregado com SUCESSO!" );
    //      display.append( "\n Aguardando..." );
          orb.run();
       }
       catch(org.omg.PortableServer.POAPackage.InvalidPolicy e)
       {      System.out.println(e);     }
       catch(org.omg.PortableServer.POAPackage.ServantAlreadyActive e)
       {      System.out.println(e);    }
       catch(org.omg.PortableServer.POAPackage.WrongPolicy e)
       {      System.out.println(e);    }
       catch(org.omg.PortableServer.POAPackage.ObjectAlreadyActive e)
       {      System.out.println(e);    }
       catch(org.omg.CORBA.PolicyError e)
       {      System.out.println(e);    }
       catch(org.omg.PortableServer.POAPackage.AdapterAlreadyExists e)
       {      System.out.println(e);    }
       catch(org.omg.CORBA.ORBPackage.InvalidName e)
       {      System.out.println(e);    }
       catch(org.omg.PortableServer.POAManagerPackage.AdapterInactive e)
       {      System.out.println(e);    }
     }//Fechamento da Classe
    }
```

2. FONTE CLIENTE.JAVA

```
import java.awt.*;
import java.awt.event.*;
class Cliente extends Frame implements ActionListener {
   private Button b1,b2,b3,b4;
   private Label l1,l2,l3,l4,l5;
   private TextField t1,t2,t3,t5;
   private String msg;
   private org.omg.CORBA.ORB orb;
   private Veiculo.Operacao op;
```

```
private String[] args;
Cliente() {
   super();
  // definição do titulo da janela
   setTitle("** Veiculos - MARCELO COSTA CAMPOS **");
   // definição do posicionamento e dimensões da janela
   setBounds(50,50,400,300);
   // definição do Layout
   setLayout(null);
   // definido a cor
   //setBackground (Color.yellow);
   setBackground(SystemColor.info);
   // criação dos controles
   b1 = new Button("Atualizacao");
   b2 = new Button("Pesquisa");
   b3 = new Button("Finalizar");
   b4 = new Button("Consulta PSS");
   l1 = new Label("Placa  :");
   l2 = new Label("Veiculo:");
   l3 = new Label("Ano    :");
   l4 = new Label("Obs    : ");
   l5 = new Label("Tipo=> BDR / BDOR");
   t1 = new TextField("");
   t2 = new TextField("");
   t3 = new TextField("");
   t5 = new TextField("");
   // definição do posicionamento e dimensões do botão
   b1.setBounds(10,230,70,25);
   b2.setBounds(110,230,70,25);
   b4.setBounds(210,230,70,25);
   b3.setBounds(320,230,70,25);
   // 1- largura inicial na tela
   // 2- posicao inicial na tela
   // 3- comprimento inicial objeto
   // 4- largura objeto
  t1.setBounds(80,50,80,25);t2.setBounds(80,80,150,25);t3.setBounds(80,110,50,25);
   t5.setBounds(310,80,50,25); l1.setBounds(20,50,50,25);l2.setBounds(20,80,50,25);
   l3.setBounds(20,110,50,25); l4.setBounds(20,140,300,25);
   l5.setBounds(265,50,150,25);
   // definição do tratador do evento tipo ActionListener
   b1.addActionListener(this);       b2.addActionListener(this);
   b3.addActionListener(this);       b4.addActionListener(this);
   // adição do botão ao frame
   add(l1); add(t1);  add(b1);  add(l2); add(t2);  add(b2);
   add(l3); add(t3);  add(b3); add(b4);     add(l4); add(t5);  add(l5);
   msg=" ";
   try {
      // Inicializa o ORB
      orb = org.omg.CORBA.ORB.init(args,null);
```

```
        //Cria instancia dos objetos para acessar as  interfaces implementadas no
op = Veiculo.OperacaoHelper.bind(orb, "poaOperacao");
      }
      catch (Exception e) {}
   }

   public void actionPerformed (ActionEvent e) {
      // verifica se o source do evento foi o botão b1
      if (e.getSource() == b1) {
         //Incluir registro
         //msg = op.Incluir(t1.getText(),t2.getText(),t3.getText());
         msg = op.Incluirpss(t1.getText(),t2.getText(),t3.getText(),t5.getText());
         l4.setText("OBS   : "+msg);       }
      if (e.getSource() == b2) {
         //Pesquisar registro
         msg = op.Pesquisar(t1.getText(),t5.getText());
         l4.setText("OBS   : "+msg);       }
      if (e.getSource() == b3) {
         // esconde e fecha a janela
         setVisible(false);
         System.exit(0);       }
      if (e.getSource() == b4) {
         //Pesquisar registro
         msg = op.Pesquisarpss(t1.getText(),t5.getText());
         l4.setText("OBS PSS: "+msg);       }
      //Limpa TextFields
      t1.setText("");  t2.setText("");   t3.setText("");
   }
   public static void main (String args[]) {
      //Cria objeto
      Cliente app = new Cliente();
      app.setVisible(true);    }
}
```

3. FONTE OPERACAO_IMPL.JAVA

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class OperacaoImpl extends Veiculo.OperacaoPOA {
  private Connection con;
  private String texto;
  private String var;
  //Estanciando o Formulario Veiculo
  PSDL       service          = new PSDL();
```

```
  Persistente servicePersistente = new Persistente();
  public OperacaoImpl()  {

     System.out.println("** Executando a Operacao Impl... **");
     Try     {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       con = DriverManager.getConnection ("jdbc:odbc:VEICULOJDBC","sa","");
     }
     catch (Exception e)     {
       System.out.println(e);     }
  }

  //DEFINIÇÃO DO MÉTODO DE INLCUSÃO DE DADOS NO CORBA
  public String Incluir( String a, String b, String c ) {
     // Inclui Registro
     try{
      Statement stmt1 = con.createStatement();
       ResultSet  rs1 = stmt1.executeQuery("SELECT * FROM VEICULO WHERE
PLACA = '"+a+"' );
       boolean more = rs1.next();
       if (more) {
         stmt1.close();
         return " ** Registro existente **";       }
       stmt1.executeUpdate("INSERT  INTO  VEICULO  VALUES  ('"+a+"',
'"+b+"','"+c+"')");
       stmt1.close();
       return " ** Registro incluido **";
     }
     catch (Exception e) {
       System.out.println(e);
       return e + "* PARE - Erro *";     }
  }

//DEFINIÇÃO DO MÉTODO DE INLCUSÃO DE DADOS NO CORBA - PSS
  public String Incluirpss( String a, String b, String c, String d ) {
     // Inclui Registro
     // Pesquisar Registro
     //==> variavel a = placa
     //==> variavel b = descricao
     //==> variavel c = ano
     //==> variavel d = banco de dados
     try{

       Statement stmt1 = con.createStatement();
       ResultSet  rs1 = stmt1.executeQuery("SELECT * FROM VEICULO WHERE
PLACA = '"+a+"' );
       boolean more = rs1.next();
       if (more) {
                  stmt1.close();
```

```
        return " ** Registro existente **";
      }
      //Pesquisando no servico de Persistencia
      texto = service.Incluirpss(a,b,c,d,"VEI");
      return " ** Registro incluido - IMPL **";      }
    catch (Exception e) {
      System.out.println(e);
      return e + "* PARE - Erro *";    }
  }

  //DEFINIÇÃO DO MÉTODO DE CONSULTA DE DADOS NO CORBA
  public String Pesquisar( String a, String b ) {

    // Pesquisar Registro
    //==> variavel a = placa
    //==> variavel b = bd
    try {
      if (b.length() <= 3)        {
        //conexao SQL
      con = DriverManager.getConnection("jdbc:odbc:VEICULOJDBC","sa","");      }
      else    {
        //conexao com o cache
        //conexao cahe
        con                    =                    DriverManager.getConnection
("jdbc:odbc:VEICULOCACHE","MCC","MCC");      }
      Statement stmt2 = con.createStatement();
      ResultSet   rs2  =  stmt2.executeQuery("SELECT  *  FROM  VEICULO  WHERE
PLACA = '"+a+"'" );
      boolean more = rs2.next();
      if (! more) {
        rs2.close();
        stmt2.close();
        return " ** Registro inexistente (4)**";     }
      texto = rs2.getString(1) + " - " + rs2.getString(2) + " - " + rs2.getString(3);
      rs2.close();
      stmt2.close();
      return texto  + " =>Registro Pesquisado (4)";      }
    catch (Exception e) {
      System.out.println(e);
      return e + "* PARE - Erro (Impl 4)*";       }
  }

  //DEFINIÇÃO DO MÉTODO DE CONSULTA DE DADOS NO CORBA PSS
  public String Pesquisarpss( String a, String b ) {

    // Pesquisar Registro
    //==> variavel a = codigo do veiculo
    //==> variavel b = banco de dados
    try {
```

```
        //Pesquisando no servico de Persistencia
        texto = service.Pesquisarteste(a,b,"VEI");
        return texto + "(PSS IMPL)";        }
    catch (Exception e) {
      System.out.println(e);
      return e + "* PARE - Erro (Impl 3)*";       }
  }

  //DEFINIÇÃO DO MÉTODO DE EXCLUSAO DE DADOS NO CORBA
  public String Excluir( String a )   {
    try       {
     Statement stmt3 = con.createStatement();
      ResultSet   rs3  =  stmt3.executeQuery("SELECT * FROM VEICULO WHERE
PLACA = '"+a+"' " );
      boolean more = rs3.next();
      if (! more) {
        stmt3.close();
        return " ** Registro Inexistente **";
      }

      stmt3.executeUpdate("DELETE FROM VEICULO WHERE PLACA = '"+ a +"'
");
      stmt3.close();
      return " ** Registro Excluido **";
    }
    catch (Exception e)       {
      System.out.println(e);
      return e + "* PARE - Erro *";     }
  }
}
```

4. PSDL.JAVA

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.sql.*;
import java.io.*;

public class PSDL extends JFrame {
   private Connection con;
   private String m_ANO;
   private String m_PLACA;
   private String m_VEICULO;
  //Estanciando o Formulario Veiculo
  Persistente  servicePersistente = new Persistente();
```

```
  public PSDL() {
     try      {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      //con   =   DriverManager.getConnection   ("jdbc:odbc:VEICULOJDBC","sa","");
}
     catch (Exception e)      {
       System.out.println("** Pare - Erro inesperado ... 1** ");      }
}
  public String Incluir( String a, String b , String c)
  {
     // Inclui Registro
     try      {
       con = DriverManager.getConnection ("jdbc:odbc:VEICULOJDBC","sa","");
            Statement stmt = con.createStatement();
       stmt.executeUpdate("INSERT INTO VEICULO VALUES ('"+a+"', '"+b+"' ,
'"+c+"')");
       stmt.close();
       return " ** Registro incluido **";
  }
     catch (Exception e) {
       return "ERRO NA INCLUSAO DE DADOS";
     }
  }

  public String Pesquisar( String a ) {
     // Pesquisar Registro
     try      {
       Statement stmt = con.createStatement();
       ResultSet   rs   = stmt.executeQuery("SELECT * FROM VEICULO WHERE
PLACA = '"+a+"' );
       boolean more   = rs.next();
       if (! more)        {
         return " ** Registro inexistente (1)**";        }
       return rs.getString(1)+rs.getString(2)  +"Registro Pesquisado";
     }
     catch (Exception e) {
       return "** PARE - Erro na Pesquisa (1)**";       }
  }

  //consulta pss
  public String Pesquisarteste( String a, String b,String classe ) {
     // Pesquisar Registro
     //==> variavel a = codigo do veiculo
     //==> variavel b = banco de dados
     try      {
       //if (b.substring(0,1) == bd.substring(0,1))
       if (b.length() <= 3)        {
         //conexao SQL
         con = DriverManager.getConnection("jdbc:odbc:VEICULOJDBC","sa","");
```

```
            }
        else        {
          //conexao com o cache
          //conexao cahe
          con                    =                    DriverManager.getConnection
("jdbc:odbc:VEICULOCACHE","MCC","MCC");
           Statement stmt = con.createStatement();
        //Chamando o metodo Select do Objeto persistente
        String query = " ";
        query = servicePersistente.select(b, classe);
        query = query + " WHERE PLACA = '"+a+"'";
        ResultSet  rst  = stmt.executeQuery(query);
        boolean more   = rst.next();
        if (! more)        {
          return  b.length() +"** Registro inexistente (PSDL)**";
        }
        return b.length() +rst.getString(1) + " - " + rst.getString(2) + " =>PSS(2)";
      }

      catch (Exception e) {
        return "** PARE - Erro na Pesquisa (2)**";     }
   }
  //incluir pss
   public String Incluirpss( String a,String b,String c,String d, String classe ) {
      // Pesquisar Registro
      // Pesquisar Registro
      //==> variavel a = placa
      //==> variavel b = descricao
      //==> variavel c = ano
      //==> variavel d = BD
      try       {
        if (d.length() <= 3)        {
          //conexao SQL
          con = DriverManager.getConnection("jdbc:odbc:VEICULOJDBC","sa","");
             }
        else        {
          //conexao com o cache
          //conexao cahe
          con                    =                    DriverManager.getConnection
("jdbc:odbc:VEICULOCACHE","MCC","MCC");
        Statement stmt = con.createStatement();
        //Chamando o metodo incluirpss do Objeto persistente
        String query = " ";
       query = servicePersistente.incluirpss(classe);
        stmt.executeUpdate("INSERT  INTO  VEICULO  (PLACA,VEICULO,ANO)
ALUES ('"+a+"', '"+b+"' , '"+c+"')");
        stmt.close();
        return " ** Registro incluido **";
```

```
    }
    catch (Exception e)      {
      return "ERRO NA INCLUSAO DE DADOS";      }
  }
}
```

5. PERSISTENTE.JAVA

```
import java.lang.reflect.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.sql.*;
public class Persistente extends JFrame
{
  //protected objectID;
  private  Connection conn;
  private  Connection conc;
  String   query = " ";
  public Persistente() {
    try {
      //Preparando as Variaveis para a Conexao com o Banco de Dados
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      //conexao cahe
      //conc                    =                    DriverManager.getConnection
("jdbc:odbc:VEICULOCACHE","MCC","MCC");
      //conexao SQL
      conn = DriverManager.getConnection("jdbc:odbc:VEICULOJDBC","sa",""); }
    catch (Exception e)      {
      System.out.println("Erro na Conexao JDBC");
      e.printStackTrace(); }
  }
  //Metodo Insercao de Dados da Classe
  public void insert () {
            //long objectID    = this.obtemProximoIdentificador();
            //String nomeClasse = this.getClasse().getName();
            String nomeClasse = "VEICULO";
            String sql       = "INSERT INTO " + nomeClasse + " ";
            //String atrib     = "(objectID,";
            String atrib     = "( ";
            String values    = " values (" ;
            try {
                  //Obtém os Atributos
                  Field[] fs = c.getDeclaredFields();
                  for(int i = 0; i < fs.length; i++)                {
                        //Obtém nome dos Atributos
```

```
                        atrib += fs[i].getName();
                        //Sé o atributo for String
                        if (fs[i].getType() == String.class)
        {
                                if (fs[i].get(this) != null)
        {
                                  values += "'" + fs[i].get(this) + "'";
                }
                                else    {
                                  values += "NULL";  }
                        }
            //Incrementando as Váriaveis
            atrib  += ", ";
            values += ", ";
            } //for
            if(atrib.length() > 0)          {
                        atrib = atrib.substring(0,atrib.length() - 1);
                        values= values.substring(0,values.length() - 1);
        }
            //Incrementando as Váriaveis
            atrib  += ")";
            values += ")";
            query = sql + atrib + values;
        stmt.executeUpdate(query);
        stmt.close();
          }
          catch (Exception se)              {
                System.out.println("Erro na Conexao JDBC");
                se.printStackTrace(); }
    }

    //Metodo Selecao de Dados -
    //1 - Recebe qual a Classe que ira implementar
    //2 - Retorna a string Query com a consulta SQL
    public String select( String bd, String classe)    {
      try      {
         //na tabela ESTRUTURA
        Statement stmt  = conn.createStatement();
        String      sql      = "SELECT CAMPO_EST FROM ESTRUTURA WHERE
ID_EST = '"+ classe +"' AND USADO_EST = 'S' ORDER BY SEQ_EST";
        String   campos = " ";
        //Recebbe o resultado da Pesquisa SQL
        ResultSet  rs  = stmt.executeQuery(sql);
        sql = " ";
        boolean more   = rs.next();
        // Fica no While ate o final dos Campos
                do {
                  campos += rs.getString(1) + ","; } while ( rs.next() );
        //Incrementando as Váriaveis
```

```
        if(campos.length() > 0)        {
                        campos = campos.substring(0,campos.length() - 1);    }
        //Agora Monta o Comando SQL para repassar pelo Return
        sql = "SELECT "+ campos +" FROM VEICULO ";
        return sql;
      }
      catch (Exception e)      {
        return "ERRO";      }
    }
    //*************************************************************//
    //*************************************************************//
    //Metodo Selecao de Dados -
    //1 - Recebe qual a Classe que ira implementar
    //2 - Retorna a string Query com a consulta SQL
    public String incluirpss( String classe)    {
      try    {
        //Consulta SQL para Levantamento dos Campos para a selecao
        //na tabela ESTRUTURA
        Statement stmt   = conn.createStatement();
        String      sql      = "SELECT CAMPO_EST FROM ESTRUTURA WHERE
ID_EST = '"+ classe +"' AND USADO_EST = 'S' ORDER BY SEQ_EST";
        String   campos = " ";
        //Recebbe o resultado da Pesquisa SQL
        ResultSet  rs  = stmt.executeQuery(sql);
        sql = " ";
        boolean more   = rs.next();
        // Fica no While ate o final dos Campos
                  do {
                     campos += rs.getString(1) + ",";
                  } while ( rs.next() );
        //Incrementando as Váriaveis
        if(campos.length() > 0)        {
                        campos = campos.substring(0,campos.length() - 1);        }
        return sql;
      }
      catch (Exception e)  {
        return "ERRO";  }
}
```