UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

DESENVOLVIMENTO DE UM SERVIÇO DE PERSISTÊNCIA TRANSPARENTE SEGUINDO O PADRÃO DE OBJETOS DISTRIBUÍDOS CORBA

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

RAFAEL CRISTIANO MACEDO

BLUMENAU, DEZEMBRO/2001

2001/2-39

DESENVOLVIMENTO DE UM SERVIÇO DE PERSISTÊNCIA TRANSPARENTE SEGUINDO O PADRÃO DE OBJETOS DISTRIBUÍDOS CORBA

RAFAEL CRISTIANO MACEDO

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

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Marcel Hugo — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Marcel Hugo

_____

Prof. Paulo Cesar Rodacki Gomes

_____

Prof. Maurício Capobianco Lopes

DEDICATÓRIA

Este trabalho é dedicado a minha mãe Relinda Rode Macedo, meu pai Antônio Carlos Macedo e a minha irmã Ana Paula Macedo, que nos piores e melhores momentos sempre me apoiaram e motivaram para a realização deste trabalho.

AGRADECIMENTOS

Agradeço principalmente a Deus por me dar sabedoria necessária durante toda a vida acadêmica e para conseguir confeccionar este trabalho. Agradeço também a minha família que em nenhum momento me deixou vacilar, em especial a minha mãe Relinda. Agradeço também aos meus amigos que me ajudaram de forma direta ou indireta para a realização deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho apresenta um estudo sobre a especificação CORBA enfocando a especificação do Serviço de Persistência e conceitos básicos sobre persistência de objetos. Apresenta também, a implementação de um Serviço de Persistência Transparente e o desenvolvimento de um protótipo de software que utiliza o Serviço de Persistência implementado.

ABSTRACT

This work introduces a research about CORBA specification, regarding to persistent service specification and basic concepts about persistent objects. This work also demonstrates a transparent persistent state service implementation and how to develop a prototype of an application using the concepts demonstrated.

1 INTRODUÇÃO

Uma das áreas da ciência da computação que apresentou um desenvolvimento rápido e acentuado durante os últimos anos foi a de redes de computadores. Segundo Queiroz (1994) este crescimento contribuiu para que os sistemas de computação fornecessem ambientes cada vez mais naturais para a evolução de sistemas distribuídos. Queiroz (1994) afirma que um sistema distribuído consiste de um conjunto de unidades de programas que cooperam entre si com o objetivo de realizar as diversas tarefas propostas pela sua definição.

Segundo Mainetti (1997) de todas as tecnologias já utilizadas no desenvolvimento de sistemas distribuídos, a orientação a objeto é a que melhor se aplica a esse modelo. Um conceito da orientação a objeto é a descentralização, assim como os sistemas distribuídos. Capeletto (1999) e Mainetti (1997) afirmam que desenvolver sistemas distribuídos utilizando a orientação a objetos facilita o processo de desenvolvimento pelo fato do sistema ser projetado sem uma preocupação com a distribuição do processamento em diferentes máquinas e processos, resolvendo assim as regras de negócio do sistema. Uma vez definidos os objetos e suas interfaces, há a preocupação com a distribuição destes objetos.

Existem vários modelos de objetos distribuídos, dentre eles destaca-se o *Distributed Component Object Model* (DCOM), da Microsoft, e o *Common Object Request Broker Architecture* (CORBA), do *Object Management Group* (OMG). O CORBA tem como principal característica a independência de sistema operacional e linguagem de programação, enquanto que o DCOM contempla somente o MS Windows NT 4.0, MS Windows 98 ES ou superior.

Fundado no fim da década de 80, o OMG, com a missão de estabelecer guias e especificações de gerência de objetos para a indústria, visando prover uma base comum para o desenvolvimento de aplicações distribuídas (OMG, 2001a), tem como sua principal contribuição o padrão CORBA. OMG (2001a) afirma que o principal objetivo do CORBA é oferecer um barramento de comunicação de software transparente entre objetos heterogêneos distribuídos.

Conforme Pires (1997) o padrão CORBA utiliza-se da colaboração entre os diferentes objetos que o constituem. Segundo Brose (2001) e OMG (2001b) estes objetos são classificados em cinco níveis. São eles:

a) *Object Request Broker* (ORB): define o barramento de objetos CORBA. É o "núcleo" do CORBA. O ORB é responsável pela transferência da informação, independentemente dos protocolos de rede e comunicação, sistema operacional, etc;
b) *Serviços Comuns*: define os objetos que contêm serviços comuns que podem ser herdados para os demais objetos, como ciclo de vida, nome, persistência, concorrência, transação, etc;
c) *Facilidades Comuns*: define as coleções de objetos que proporcionam serviços comuns para os objetos de aplicações;
d) *Interfaces de Domínio*: define as coleções de interfaces de domínio específico para aplicações de áreas específicas como finanças, transportes, etc.
e) *Interfaces de Aplicações*: define os objetos específicos para as aplicações voltadas ao usuário final.

Pelo fato do padrão CORBA ser não-proprietário (OMG, 2001a), existem atualmente no mercado várias implementações do ORB que podem ser adquiridas para serem utilizadas na implementação de um sistema distribuído. Mas os serviços oferecidos por elas podem não ser satisfatórios para uma determinada aplicação. Nesta situação pode-se implementar um serviço que atenda as necessidades específicas da aplicação, substituindo ou acrescentando o serviço ao ORB, respeitando a especificação do OMG para o determinado serviço.

Como na maioria das implementações do ORB consultadas os serviços básicos já são implementados, verifica-se que o grande diferencial para a utilização do CORBA poderá estar nos serviços de persistência, concorrência, transação, segurança e pesquisa.

Pelo fato do OMG ter formalmente especificado o serviço de persistência e nesta especificação não constar como implementar o serviço, mas sim o que o serviço deve contemplar, ela permite inúmeras formas de implementação. Assim, pretende-se projetar e implementar o serviço de persistência, integrando-o a um ORB já existente no mercado e executar um protótipo de aplicação demonstrando o uso do serviço implementado integrado ao ORB.

O serviço de persistência consiste em armazenar fisicamente um objeto, de modo que, os objetos continuem a existir após a execução dos programas que os criaram ou atualizaram. Estes objetos podem ser armazenados de várias formas, por exemplo, em um banco de dados orientado a objetos ou banco de dados objeto-relacional ou em um conjunto de arquivos (OMG, 1999).

Será especificado um protótipo de aplicação através da UML para demonstrar o funcionamento da implementação ORB selecionada. O protótipo será composto por um servidor que será implementado utilizando a linguagem de programação Java, e clientes que serão implementados utilizando Java.

1.1 OBJETIVOS DO TRABALHO

O objetivo principal do trabalho é projetar e implementar a persistência transparente seguindo a especificação do serviço de persistência do padrão CORBA.

Os objetivos específicos do trabalho são:

a) selecionar um ORB existente no mercado para poder integrar o serviço de persistência;
b) projetar e implementar uma aplicação distribuída utilizando a especificação do OMG para o serviço de persistência; e
c) verificar a flexibilidade e a interoperabilidade do CORBA e do serviço de persistência implementado.

1.2 ESTRUTURA DO TRABALHO

O trabalho foi dividido em 6 capítulos, descritos a seguir.

O primeiro capítulo define os objetivos do trabalho, apresentando a justificativa para o seu desenvolvimento.

O segundo capítulo apresenta uma visão geral sobre sistemas distribuídos. Neste capítulo também são demonstrados os conceitos de orientação a objeto, bem como a união destas duas tecnologias.

O terceiro capítulo demonstra o padrão CORBA, sua história, arquitetura, conceitos e serviços oferecidos.

O quarto capítulo aborda o serviço de persistência definido pelo padrão CORBA e conceitos sobre persistência de objetos.

O quinto capítulo demonstra o desenvolvimento do serviço de persistência, e do protótipo de aplicação que será usado para validar o serviço de persistência desenvolvido.

O sexto capítulo apresenta as sugestões, dificuldades encontradas e conclusões do trabalho.

2 OBJETOS DISTRIBUÍDOS

Os sistemas de objetos distribuídos provêm do desenvolvimento de duas tecnologias da computação: os sistemas distribuídos e a orientação a objetos (Siegel, 1996). Os sistemas distribuídos surgiram com a necessidade de interligar vários sistemas computacionais separados geograficamente através de redes de computadores. Ao mesmo tempo, um outro paradigma era desenvolvido, a orientação a objetos que tenta modelar e representar o mundo real da forma que é realmente.

2.1 SISTEMAS DISTRIBUÍDOS

Segundo Capeletto (1999), os sistemas distribuídos apresentam um rápido crescimento devido ao aumento das redes de computadores e da velocidade de transmissão de dados entre estas redes. Uma vez que a distribuição implica, de um modo geral, na separação física entre os componentes da aplicação, os aspectos de transmissão de dados desempenham um papel fundamental nos sistemas distribuídos.

A principal característica de um sistema distribuído é a sua composição. Um sistema distribuído é formado por várias aplicações componentes que cooperam dinamicamente entre si. Dessa forma, outras características são herdadas: disponibilidade, desempenho, autonomia, assincronismo, dependência, mobilidade, afastamento, concorrência, heterogeneidade, falta de estado global, evolução, ocorrência de falhas parciais, entre outras.

Quanto às necessidades computacionais de usuários, observa-se uma tendência de demanda crescente de sistemas distribuídos, tanto para novas aplicações como para a substituição de aplicações existentes, baseadas em sistemas centralizados. O conceito de “servidor”, predominante ainda hoje, onde poucos computadores realizam todas as tarefas, traz consigo a prática de que os usuários devem ir ao servidor, ao invés do servidor ir ao usuário. Existe uma variedade de aplicações de naturezas comercial, industrial e científica, as quais grande parte do processamento pode ser executada eficientemente nos locais onde os dados são gerados.

2.1.1 FACILIDADES E DIFICULDADES DOS SISTEMAS DISTRIBUÍDOS

Um sistema distribuído apresenta uma série de vantagens. Entre elas, destaca-se a possibilidade de se obter alto desempenho, alta confiabilidade, compartilhamento efetivo de recursos, extensibilidade, redução de custos, entre outras.

Uma aplicação distribuída é, intrinsecamente, mais complexa do que a sua correspondente local. As seguintes dificuldades podem ser observadas em sistemas distribuídos: a heterogeneidade de aplicações, o assincronismo, a concorrência, a ocorrência de falhas parciais, segurança na rede, gerenciamento de transações. Essas dificuldades também podem ser encontradas em outras arquiteturas, como a arquitetura cliente-servidor, porém, em sistemas distribuídos essas dificuldades podem ser maiores.

2.2 ORIENTAÇÃO A OBJETO

Conforme Coad (1992), uma das maiores preocupações da análise de sistemas é com a compreensão do domínio do problema, evolução contínua, comunicação dos fatos e reutilização. Para conseguir isso, o analista, na fase de entrevistas com o usuário, deve obter o máximo de informações possíveis sobre o domínio do problema e ter consciência que as pessoas envolvidas com o sistema é que serão responsáveis para se chegar à compreensão do domínio do problema.

A indústria de software percebendo a necessidade de métodos para representação de sistemas cada vez maiores e complexos que os de alguns anos atrás, direcionou suas pesquisas para o desenvolvimento de uma metodologia baseada nos conceitos do paradigma da orientação a objeto.

As linguagens e ferramentas orientadas a objeto permitem que os problemas tecnológicos do mundo real, tais como ambientes complexos de escritório ou vários problemas de engenharia, sejam expressos mais facilmente e naturalmente usando-se componentes modularizados. Sistemas orientados a objeto tentam tornar a programação mais direta e objetiva (Martin, 1994).

Segundo Coad (1992), a orientação a objeto é baseada em três conceitos básicos que estão intrínsecos na maneira de pensar dos seres humanos. São eles:

a) diferenciação, baseado na experiência de cada um, de objetos particulares e seus atributos. Por exemplo, quando classifica-se que um objeto é uma árvore e que possui atributos como altura, peso, idade;
b) distinção entre objetos como um todo e entre suas partes componentes. Por exemplo, quando separa uma árvore de seus galhos; e
c) formação de, e distinção entre, as diferentes classes de objetos. Por exemplo, quando formam uma classe de todas as árvores, uma outra classe de todas as rochas e distinguem-nas.

Segundo Martin (1994), o projeto orientado a objeto tem como objetivo estimular reuso, manutenção e estabilidade, objetivos que não são replicados por nenhuma etapa da análise de requisitos. Durante a análise, os objetos são escolhidos de forma a representar entidades do mundo real, de forma a otimizar a compreensão do problema.

2.2.1 CONCEITOS OO

Segundo Coad (1992), a orientação a objeto baseia-se na aplicação uniforme dos princípios para administração da complexidade de um domínio de problemas e das responsabilidades do sistema dentro dele. Esses princípios, utilizados nos diferentes métodos de análise e projeto orientado a objeto, são apresentados a seguir:

a) **Abstração**: é a habilidade de ignorar os aspectos de um assunto não relevantes para o propósito em questão, tornando possível uma concentração maior nos assuntos principais. A abstração consiste portanto na seleção que um desenvolvedor faz de alguns aspectos, suprimindo outros;
b) **Encapsulamento**: estrutura utilizada para ocultar informações. Essa restrição visa obter melhor legibilidade, manutenibilidade e principalmente reutilização no desenvolvimento de um novo sistema;
c) **Classe e Objeto**: das idéias de abstração e encapsulamento tem-se o Tipo Abstrato de Dados (TAD), que possui uma interface para ocultar detalhes de implementação, possibilitando que se tenham diferentes especificações, em diferentes tempos, sem afetar as aplicações que utilizam o TAD. Uma classe é um TAD. Uma classe é

portanto um conjunto de objetos similares. Uma instância de uma classe é chamada objeto da classe;

d) **Herança**: é o mecanismo para expressar a similaridade entre classes, simplificando a definição de classes iguais a outras que já foram definidas. Representa generalização e especialização, tornando explícitos os atributos e serviços comuns em uma hierarquia de classe. A herança permite, portanto, a reutilização de especificações comuns. O reconhecimento da similaridade entre classes forma uma hierarquia de classes, onde superclasses representam abstrações generalizadas e subclasses representam abstrações, onde atributos e serviços específicos são adicionados, modificados ou removidos;
e) **Agregação**: também conhecido com Todo-Parte, é um mecanismo que permite a construção de uma classe agregada a partir de outras classes componentes. Os objetos da classe agregada (Todo) têm objetos das classes componentes (Parte);
f) **Associação**: é uma união ou conexão de idéias. A associação é modelada através de uma conexão de ocorrências. Uma conexão de ocorrência é um modelo de mapeamentos de domínio de problemas que um objeto precisa ter com outros objetos, para cumprir suas responsabilidades;
g) **Polimorfismo**: significa sobrecarga. Um objeto pode ter várias formas. O mesmo método pode atuar de diversos modos em classes diferentes.

2.2.2 UML

Segundo Booch (2000) as primeiras linguagens de modelagem orientadas a objetos surgiram entre as décadas de 1970 e 1980, chegando a pouco mais de 10 linguagens. Até 1994 haviam mais de 50 linguagens de modelagem orientada a objetos e nenhuma linguagem era capaz de atender totalmente as necessidades de modelagem orientadas a objeto. Assim, os idealizadores das linguagens de maior destaque no mercado reuniram-se para formar uma linguagem que unificasse os pontos fortes dos três métodos mais reconhecidos mundialmente, que são os métodos de Booch, o *Object-Oriented Software Engineering* (OOSE) de Jacobson, e o *Object Modeling Technique* (OMT) de Rumbaugh, surgindo assim a *Unified Modeling Language* (UML) em 1996.

Booch (2000) afirma que a UML é uma linguagem de modelagem orientada a objeto que pode ser empregada para a visualização, a especificação, a construção e a documentação de sistemas orientados a objeto cuja abrangência pode incluir sistemas de informação corporativos a serem distribuídos e até sistemas complexos embutidos de tempo real.

Conforme OMG (2001a), a UML utiliza doze diagramas para a modelagem de sistemas. Os diagramas são classificados como diagramas estáticos, dinâmicos e funcionais.

Os diagramas estáticos descrevem os dados e a estrutura de objetos da aplicação. Os diagramas estáticos proporcionam a estrutura necessária na qual podem ser modelados os diagramas dinâmicos e funcionais.

Os diagramas dinâmicos descrevem os aspectos de um sistema relacionados ao tempo e a seqüência de operações, a interação entre os objetos da aplicação, os eventos e as mudanças de estado dos objetos.

Os diagramas funcionais descrevem a configuração, a disposição, a funcionalidade, a instalação e o gerenciamento dos objetos da aplicação.

Os diagramas da UML podem ser classificados conforme o Quadro 1:

Quadro 1: Classificação dos Diagramas da UML

Diagrama	Tipo
Diagrama de Classe	Estático
Diagrama de Objeto	Estático
Diagrama de Componente	Estático
Diagrama de Implantação	Estático
Diagrama de Casos de Uso	Dinâmico
Diagrama de Estado	Dinâmico
Diagrama de Seqüência	Dinâmico
Diagrama de Colaboração	Dinâmico
Diagrama de Atividade	Dinâmico
Diagrama de Pacote	Funcional
Diagrama de Modelo	Funcional
Diagrama de Subsistemas	Funcional

2.3 CONCEITOS SOBRE OBJETOS DISTRIBUÍDOS

Um objeto distribuído é essencialmente um componente, uma peça de software com inteligência autocontida, que pode interoperar com outros objetos distribuídos através de sistemas operacionais, redes, linguagens, aplicações, ferramentas e equipamentos diversos. (Montez, 1997).

Diversos modelos e arquiteturas distribuídas têm sido desenvolvidos. A característica de heterogeneidade impõe a necessidade de especificações abertas, com interfaces padronizadas e públicas, levando ao desenvolvimento de *middlewares* abertos.

Segundo Capeletto (1999), o *middleware* é uma infra-estrutura comum utilizada para integrar as aplicações distribuídas em uma rede de computadores. Funciona como um barramento de comunicação comum a todas as aplicações, simplificando a troca de informações. Seu principal objetivo é tornar transparente para as aplicações a dificuldade de se trabalhar com a heterogeneidade dos protocolos de rede dos diversos sistemas operacionais nos quais os sistemas distribuídos são executados. Esse elemento resolve o problema de integração das aplicações sendo executadas na rede e deve garantir assim que aplicações implementadas, utilizando-se diferentes linguagens, possam trocar informações de forma completamente transparente.

O conceito de sistemas abertos aborda um significante leque de tecnologias e especificações, envolvendo a questão de interoperabilidade de uma variedade de sistemas baseados em padrões formais. De forma diversa aos ambientes proprietários, ambientes de sistemas abertos permitem aos usuários escolherem de um vasto grupo de computadores e tecnologias, que se adequam a suas necessidades. Esses sistemas oferecem também uma crescente interoperabilidade, escalabilidade e portabilidade, permitindo a sistemas diferentes conviverem juntos, e softwares desenvolvidos em uma plataforma poderem executar em outra plataforma com adaptações mínimas (Siegel, 1996).

2.4 OMG

O *Object Management Group* (OMG) é um consórcio de empresas envolvidas com computação de todo o mundo que se reúnem para estabelecer padrões e especificações para aplicações orientadas a objeto (OMG, 2001a).

O OMG estabeleceu a arquitetura CORBA como uma forma de especificar um *middleware* aberto composto de objetos distribuídos. CORBA define um tipo de "barramento de software" que permite que componentes de software sejam conectados formando um sistema coeso. Os conceitos utilizados nos componentes de software são similares aos dos componentes de hardware, e são adotados no CORBA utilizando-se a orientação a objeto.

Todos os padrões e especificações estão disponíveis gratuitamente na Internet para todas as empresas ou pessoas que queiram utilizá-las, independentemente de serem membros ou não do OMG.

3 CORBA

Segundo Pires (1997), o *Common Object Request Broker Architecture* (CORBA) foi desenvolvido inicialmente em 1989 pelo OMG, reunindo um fórum com mais de 700 empresas, dentre elas, nomes como HP, IBM, Digital e Sun.

O intuito do padrão do CORBA é oferecer uma arquitetura aberta, não proprietária, capaz de ser uma via única de comunicação disponível a qualquer desenvolvedor.

O padrão CORBA é uma abordagem para objetos distribuídos que permite aos objetos invocarem métodos em objetos distribuídos em redes de computadores, independentemente da localização do objeto, do sistema operacional que o objeto executa e da linguagem que o objeto é implementado (Siegel, 1996).

A implementação do CORBA resulta em produtos encontrados no mercado denominados de *Object Request Broker* (ORB). O ORB habilita aos objetos enviarem e receberem requisições e, da mesma maneira, receberem respostas a suas requisições, de forma transparente em um sistema distribuído. O ORB é a fundação para se construir aplicações, utilizando objetos distribuídos, com características de interoperabilidade entre aplicações em ambientes heterogêneos ou homogêneos (Montez, 1997).

Siegel (1996) afirma que o CORBA é baseado, dentre outros aspectos, na construção e armazenamento de interfaces para os objetos. Estas interfaces são armazenadas em repositórios de interfaces. No repositório de interface os objetos podem fazer requerimentos para obter informações sobre determinada interface, ou mesmo criar uma nova interface no repositório.

As interfaces devem ser construídas para serem utilizadas no instanciamento de objetos, para controlarem o acesso aos objetos e controlar o relacionamento entre estilos de objetos (gerenciamento de classes). A padronização da arquitetura da criação de tais interfaces possibilita a criação de aplicações que fazem instanciamento de objetos e manipulação de requerimentos das operações contidas nos mesmos.

Um desenvolvedor pode criar tais interfaces que podem ser usadas por qualquer ORB num sistema. Desta forma, fica garantido que um objeto poderá ser instanciado com a

interface em questão pelo ORB, e o cliente que precisar utilizar o objeto reconhecerá, pela interface, as operações que espera encontrar - caso tenha conhecimento desta interface (Invocação Estática). Pode, ainda recorrer ao repositório para saber quais operações são fornecidas por esta interface (Invocação Dinâmica).

3.1 OBJECT MANAGEMENT ARCHITECTURE (OMA)

A *Object Management Architecture* (OMA) é um conjunto de conceitos e padrões para a especificação de objetos que suportam aplicações distribuídas. Segundo Brose (2001), a OMA provém de dois modelos fundamentais: o Modelo de Objetos (*Object Model*) e o Modelo de Referência (*Reference Model*).

Todos os padrões e especificações do OMG baseiam-se nos conceitos e padrões definidos pela OMA. Mas OMG (1997) afirma que os conceitos e padrões contidos na OMA são apenas de caráter sugestivo.

3.1.1 MODELO DE OBJETOS

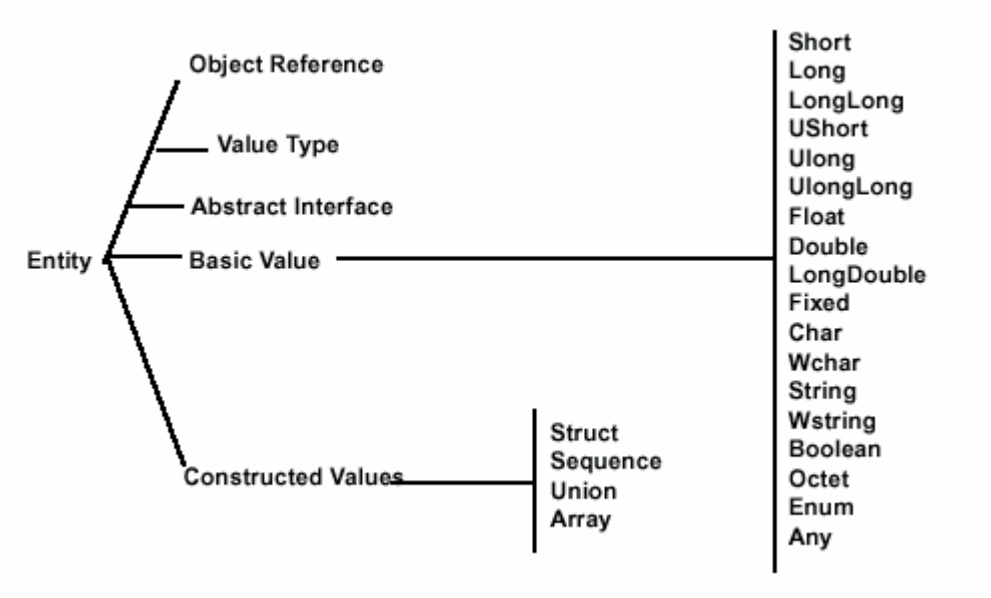
Segundo OMG (1997), o Modelo de Objetos contém conceitos e terminologias de objetos que definem um modelo parcial com características chave de objeto.

Um conceito básico é o de sistema de objetos que é composto por entidades denominadas objetos. Um objeto é uma entidade identificável que fornece serviços aos clientes. Um cliente de um serviço é qualquer entidade capaz de requisitar serviços através de eventos denominados requisições. Uma requisição possui informação associada que consiste basicamente da operação, do objeto destino, dos parâmetros e do contexto da requisição.

Objetos podem ser criados e destruídos através de determinadas requisições. O resultado de uma criação de objeto é revelado ao cliente na forma de uma referência de objeto que denota o novo objeto.

Um valor é uma instância de um tipo de dados definido que pode ser usado como parâmetro em uma requisição. Um valor que identifica um objeto é denominado nome de objeto, e uma referência de objeto é um nome de objeto que denota um objeto em particular. Na Figura 1, estão os valores legais definidos no Modelo de Objeto pelo OMG.

Figura 1: Valores legais do Modelo de Objeto



Fonte: OMG (2001a)

Uma requisição pode ter parâmetros que podem ser de entrada, saída, ou de entrada e saída, e que são identificados pelas suas posições na requisição. Ela pode ter também um contexto que fornece informação adicional sobre a própria requisição. Uma requisição pode retornar um valor de resultado para os clientes, além de retornar os parâmetros de saída. Entretanto, se uma condição anormal ocorrer, uma exceção é gerada.

Uma interface é uma descrição de um possível conjunto de operações que um cliente pode requisitar de um objeto. Diz-se que um objeto satisfaz uma interface se ele pode ser especificado como objeto destino em cada operação descrita pela interface. Uma herança de interface fornece o mecanismo de composição para permitir um objeto suportar interfaces múltiplas.

Uma interface pode ter atributos. Um atributo é logicamente equivalente a declaração de duas operações; uma para recuperar o valor do atributo e uma para alterar o valor do atributo. Um atributo pode ser *read-only*, neste caso somente a operação para recuperar o valor do atributo é definida.

Uma operação é uma entidade que denota um serviço que pode ser requisitado. Uma operação possui uma assinatura que, de um modo geral, descreve os valores válidos dos

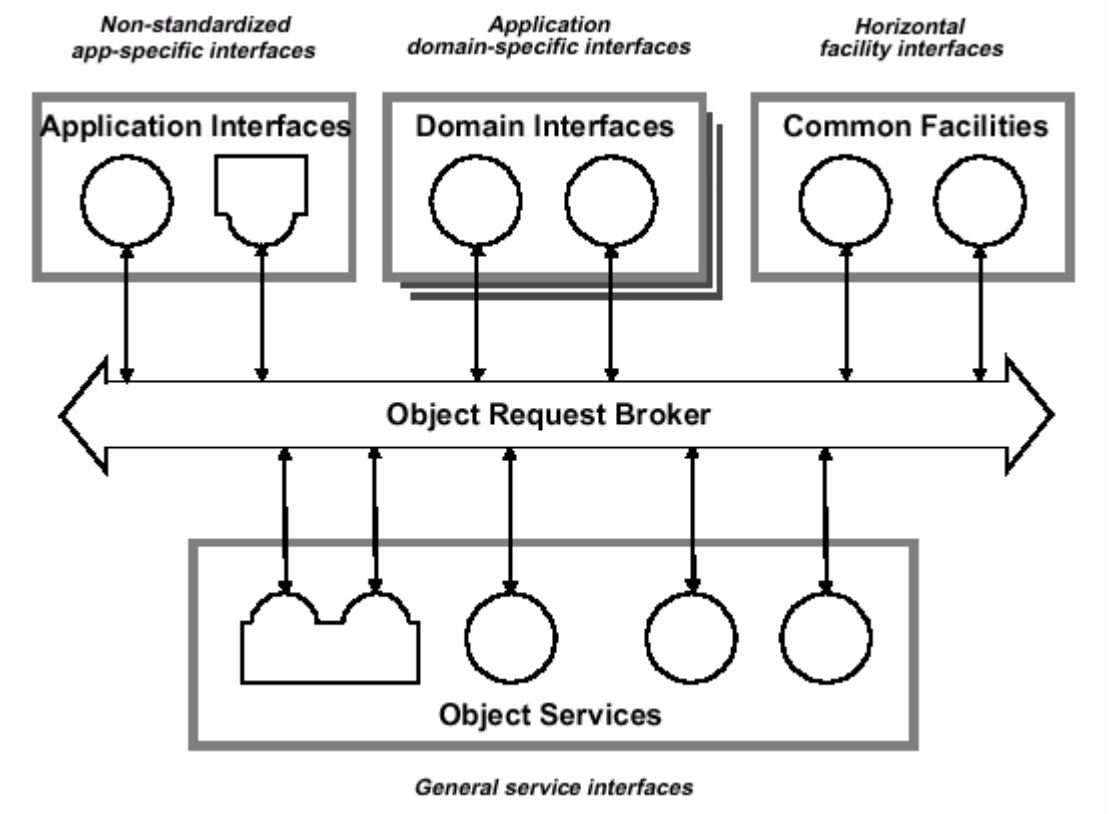
parâmetros, dos resultados retornados da requisição, a exceção definida pelo usuário que pode ser sinalizada para terminar uma requisição de operação, e a informação de contexto que será fornecida à implementação do objeto.

Na implementação de objeto, um método é o código que é executado para fornecer o serviço e a execução de um método é denominada ativação do método.

3.1.2 MODELO DE REFERÊNCIA

Conforme OMG (1997), o Modelo de Referência identifica e caracteriza os componentes, interfaces e protocolos que compõem a OMA. O Modelo de Referência é uma estrutura para a padronização de interfaces e serviços que as aplicações possam utilizar. É constituído por cinco componentes: ORB, Objetos de Serviço, Facilidades Comuns, Interfaces de Domínio e Interfaces de Aplicação. A Figura 2 representa o relacionamento entre os componentes.

Figura 2: Modelo de Referência



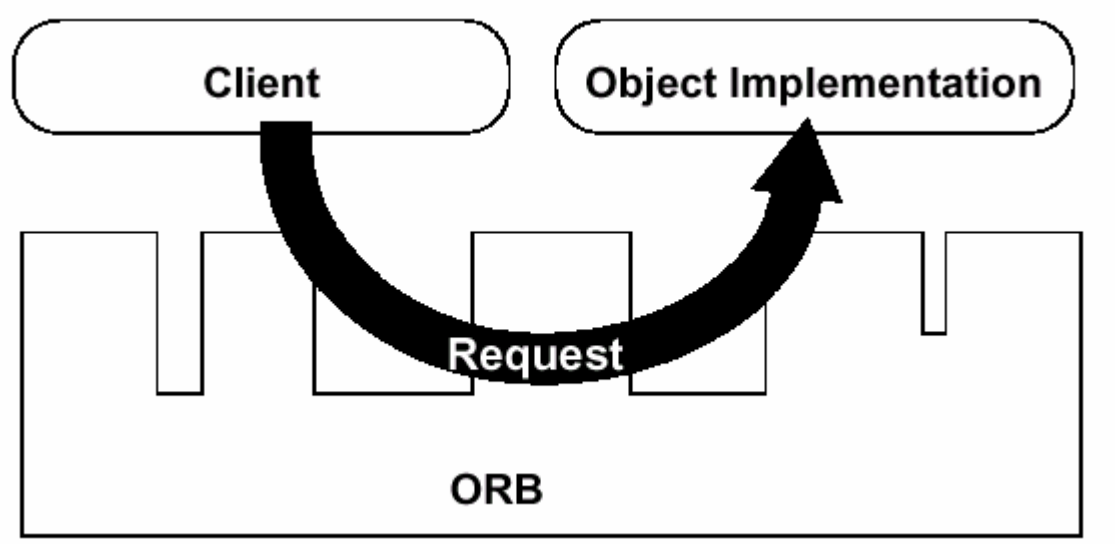
Fonte: OMG (1997)

Ao contrário do Modelo de Objetos, o Modelo de Referência é aplicado diretamente ao padrão CORBA, identificando a sua estrutura e como os objetos são organizados internamente.

3.1.2.1 ORB

Objetos clientes requisitam serviços às implementações de objetos através de um ORB. O ORB é responsável por todos os mecanismos requeridos para encontrar o objeto, preparar a implementação do objeto para receber a requisição, e executar a requisição. Assim, o cliente vê a requisição de forma independente de onde o objeto está localizado, qual linguagem de programação ele foi implementado, ou qualquer outro aspecto que não está refletido na interface do objeto (Montez, 1997). A Figura 3 demonstra como uma requisição é feita do cliente para o servidor, denominado também de implementação do objeto.

Figura 3: Requisição através do ORB



Fonte: OMG (2001a)

3.1.2.2 SERVIÇOS COMUNS

Segundo Brose (2001), um conjunto de especificações de interface disponibilizam os serviços fundamentais que os desenvolvedores necessitam para gerenciar seus objetos e executar operações complexas. Estas interfaces são denominadas Serviços Comuns.

Conforme OMG (2001b), na versão 2.5 do CORBA, existem 15 serviços especificados pela OMG. São eles:

a) *Collection Service*: Este serviço proporciona a habilidade de trabalhar com múltiplos objetos como se fossem um único objeto, permitindo que as consultas retornem coleções de objetos bem como um único objeto. Por exemplo, se um objeto aluno participou de vários seminários durante o período em que esteve na universidade, é possível lidar com estes seminários como se fossem um único objeto ao invés de ter que trabalhar com cada um em separado. Isto permite escrever com mais facilidade o código necessário para emitir um boletim, pois seria possível percorrer rapidamente a coleção de seminários pedindo que eles emitam suas informações da forma apropriada.
b) *Persistent Object Service*: Este serviço proporciona uma interface única para armazenar persistentemente objetos de uma variedade imensa de mecanismos, incluindo os banco de dados orientados a objeto, banco de dados relacionais e arquivos.
c) *Concurrency Service*: Este serviço proporciona um gerente de bloqueio para suportar a concorrência de objeto, permitindo que vários usuários trabalhem com o mesmo objeto simultaneamente sem que um sobrescreva o trabalho do outro.
d) *Property Service*: Este serviço proporciona operações que permitem que a associação de propriedades com qualquer objeto que esteja acima e além do conjunto atual de atributos.
e) *Query Service*: Este serviço proporciona as funções básicas de consulta de objetos e é baseado na especificação ANSI SIL3 e no padrão *Object Query Language* (OQL).
f) *Event Service*: Este serviço permite que os objetos registrem dinamicamente seu interesse ou a falta de interesse em eventos específicos. Por exemplo, quando um objeto relatório estiver se auto-imprimindo ele indicará ao ORB que está interessado em qualquer erro de impressão que aconteça, para que possa redirecioná-lo para outro dispositivo de saída no caso de ocorrer algum problema. Uma vez que a impressão esteja completa, o objeto relatório informará ao ORB que não está mais interessado na situação da impressora. A vantagem disto é que a impressora simplesmente precisa informar ao ORB que algo está errado para que o ORB possa informar a qualquer objeto interessado.

g) *Relationship Service*: Este serviço proporciona uma maneira dinâmica de criar associações entre objetos que não sabem nada um do outro, assim como um mecanismo para atravessar estas associações. Este serviço pode ser utilizado para manter a integridade referencial entre os objetos e para pôr em vigor os relacionamentos de agregação.

h) *Externalization Service*: Este serviço proporciona uma abordagem-padrão para se colocar e retirar dados de um objeto através de um processo chamado *Object Streaming*. Os objetos precisam com freqüência ser transferidos através de uma conexão que não seja de rede – talvez o objeto precisa ser enviado para um computador remoto via modem. Pelo fato de somente transferir dados é preciso um mecanismo que converta objetos em dados, e dados em objetos para que assim possam ser transmitidos entre computadores/redes diferentes. Este processo de conversão é chamado de *Object Streaming*.

i) *Security Service*: Este serviço proporciona uma maneira de proteger os servidores dos clientes gerenciando os direitos de acesso e proporcionando a autenticação dos objetos, assim como definindo os esquemas de criptogramas e assinaturas eletrônicas. Os objetos devem apenas esperar receber mensagens de objetos que têm em primeiro lugar permissão para enviar mensagens. Portanto deve existir um mecanismo de segurança disponível para definir o que é permitido que cada objeto da aplicação pode executar.

j) *Naming Service*: Este serviço permite que os objetos se localizem uns aos outros pelo nome, independentemente de onde estejam na rede. O serviço de nomes basicamente atua como o catálogo de páginas amarelas, associando o nome de um objeto com seu endereço na rede. Por exemplo, um usuário trabalhando com uma tela de registro de um seminário pode querer inscrever “José Pereira” para o seminário de “Mecânica” de código MEC018. o objeto tela deveria enviar para o ORB a mensagem “Seminário ‘MEC018’ adicionar seminarista ‘José Pereira’”. O ORB utiliza o serviço de nomes para localizar tanto o objeto seminarista quanto o objeto seminário, enviando-lhes as mensagens apropriadas. O bom disto é que a tela de registro não precisa saber onde na rede estão os objetos de negócios que interagem com ela. O ORB encapsula este conhecimento.

k) *Time Service*: Este serviço permite sincronizar os relógios de todos os computadores, assim parece que toda a rede está operando ao mesmo tempo. Isto é importante porque é possível trabalhar com o mesmo objeto simultaneamente em várias máquinas, assim é necessário identificar a ordem na qual qualquer mudança deve ser aplicada ao objeto. Sem um relógio que marque o mesmo horário isto é impossível.

l) *Licensing Service*: Este serviço proporciona a capacidade de medir o uso de objetos para assegurar uma compensação justa pelo seu uso, tornando possível "o aluguel" de softwares para as pessoas por períodos específicos ou por uso. Talvez uma pessoa precise utilizar um programa de desenho sofisticado para uma apresentação a fazer, mas não compensa a despesa de comprá-lo no momento. Pelo fato de precisar dele somente para apresentação, decide alugá-lo por uma parcela do preço do pacote inteiro. O serviço de licença informa ao autor do software a quantidade utilizada para que possa ser cobrado corretamente.

m) *Trading Object Service*: Este serviço proporciona uma indicação do que um objeto é capaz de fazer, permitindo que outros objetos descubram que mensagens podem ser-lhes enviadas, assim como que parâmetros precisam enviar. Este é basicamente um serviço de busca com conceito similar ao catálogo de páginas amarelas – os objetos anunciam o que podem fazer, e os outros objetos utilizam este serviço de propaganda para encontrar o objeto que proporcione o serviço que atenda às suas necessidades.

n) *Life Cycle Service*: Este serviço define convenções e operações para criar, mover e apagar os objetos CORBA. As convenções permitem aos objetos clientes executar as operações do serviço de ciclo de vida em objetos com localizações diferentes.

o) *Transaction Service*: Este serviço fornece coordenação de um compromisso de duas fases entre objetos recuperáveis utilizando transações planas ou cascateadas, tornando possível escrever, simultaneamente, diversos objetos nos seus respectivos locais de persistência. Este serviço ajuda a manter a integridade dos objetos para que sejam consistentes uns com os outros. Por exemplo, quando um pedido é efetuado por um cliente os itens daquele pedido devem ser enviados. Quando um item do inventário é removido para atender ao pedido, é necessário certificar-se de que o atributo "estoque" do objeto "item do inventário" e o objeto "item do pedido"

> sejam criados ao mesmo tempo – se o item do inventário for atualizado mas o item do pedido não for criado corretamente, então o item de inventário não refletirá, de forma correta, o nível de estoque atual (o item ainda permanecerá no estoque porque não será remetido). Um problema similar ocorre quando o objeto item do inventário não é atualizado, mas o objeto item do pedido é criado. Os objetos do sistema devem permanecer consistentes uns com os outros; isto é facilitado pelo serviço de transação.

Wutka (1997) cita que a grande vantagem do CORBA é que todos os serviços estão disponíveis através da interface ORB selecionada; assim, a aplicação que utilizar o ORB terá estes serviços disponíveis. Mas Wutka (1997) afirma ainda que nem todos os serviços especificados pelo OMG podem estar implementados no ORB. Por exemplo, o Visibroker 4.5, da Inprise tem disponível apenas os serviços de Nome e Eventos (Inprise, 2000). Outro exemplo, o Orbix 2000 1.2.1, da Iona tem disponíveis os serviços de Nome, Eventos, Localização e Ativação (Iona, 2001). Desta maneira, os serviços podem ser adicionados ou retirados de uma interface ORB.

Os serviços de persistência e segurança são muito individuais para cada aplicação, o que proporciona um desejo maior de “personalização” destes serviços em particular. Por exemplo, determinada aplicação necessita de um fator maior de segurança do que o padrão. Outro exemplo pode ser uma aplicação armazenada em um Banco de Dados e outra em arquivos.

3.1.2.3 FACILIDADES COMUNS

Conforme Capeletto (1999), é uma coleção de serviços que todas as aplicações podem compartilhar, mas que não são tão fundamentais como os Objetos de Serviço. Segundo Brose (2001), algumas facilidades comuns estão especificadas completamente, como as facilidades de gerenciamento de sistemas, de internacionalização e tempo, troca de dados, agentes móveis e facilidade de impressão. Por exemplo, uma empresa deseja que todas as suas aplicações distribuídas tenham a mesma interface com o usuário. Para conseguir esta característica, o desenvolvedor deve utilizar a facilidade comum de gerenciamento de sistemas.

3.1.2.4 INTERFACES DE DOMÍNIOS

O OMG contém um grande número de grupos de interesse e forças-tarefa que focam um domínio de aplicação em particular, como telecomunicações, internet, finanças, transportes, medicina (Brose, 2001). Esta área de padronização foi separada das Facilidades Comuns em 1996, onde, até então, era denominada de Facilidades Verticais.

Estes grupos de interesse e forças tarefas tentam especificar e padronizar todas as características pertinentes somente à área de interesse em que mantêm seus esforços concentrados.

3.1.2.5 INTERFACES DE APLICAÇÕES

As interfaces de aplicações representam todos os objetos de negócio de uma aplicação. Estes objetos são completamente de alto-nível que têm funcionalidade de interesse direto do usuário final. Conforme Capeletto (1999), as interfaces de aplicações correspondem a tradicional noção de aplicação, portanto elas não são padronizadas pelo OMG.

3.2 INTERFACE DEFINITION LANGUAGE (IDL)

Brose (2001) afirma que a OMG *Interface Definition Language* (IDL) é uma linguagem declarativa para definir as interfaces dos objetos. O CORBA utiliza a IDL como uma forma de especificar um contrato entre os objetos. Pelo fato de ser uma linguagem declarativa, garante que os componentes em CORBA sejam autodocumentáveis, permitindo que diferentes objetos, escritos em diferentes linguagens, possam interoperar através das redes e de sistemas operacionais (Montez, 1997).

Uma definição de interface escrita em OMG IDL define completamente a interface e especifica cada parâmetro das operações. É importante ressaltar que os objetos não são escritos em OMG IDL, que é uma linguagem puramente descritiva. Eles são escritos em linguagens que possuem mapeamentos definidos dos conceitos existentes em OMG IDL. Segundo OMG (2001b), algumas linguagens mapeadas são: Ada, Java, C, C++, Lisp, Cobol, Python e Smalltalk. O Quadro 2 exemplifica uma definição IDL.

Quadro 2: Exemplo de Definição IDL

```
module Banco
{
  struct Conta
  {
    private string nome;
    public string endereco;
    private float saldo;
    void debito(in float valor);
    void credito(in float valor);
    factory init(in string nome, in string endereco, in float saldo);
  };

  interface Bancario
  {
    Conta cria_conta(in string nome, in string endereco, in float saldo);
  };
};
```

Todo os produtos ORB têm a característica de compilar uma definição IDL (OMG, 2001b). Ao compilar uma definição IDL, vários arquivos são gerados com base na definição IDL. Estes arquivos são gerados na linguagem em que o produto ORB está mapeado. Definido como padrão pelo OMG (2001b), ao compilar o arquivo IDL "Classe.idl", por exemplo, em um produto ORB com mapeamento da IDL para a linguagem Java, os seguintes arquivos devem ser gerado, o conteúdo destes arquivos são detalhados nas próximas seções:

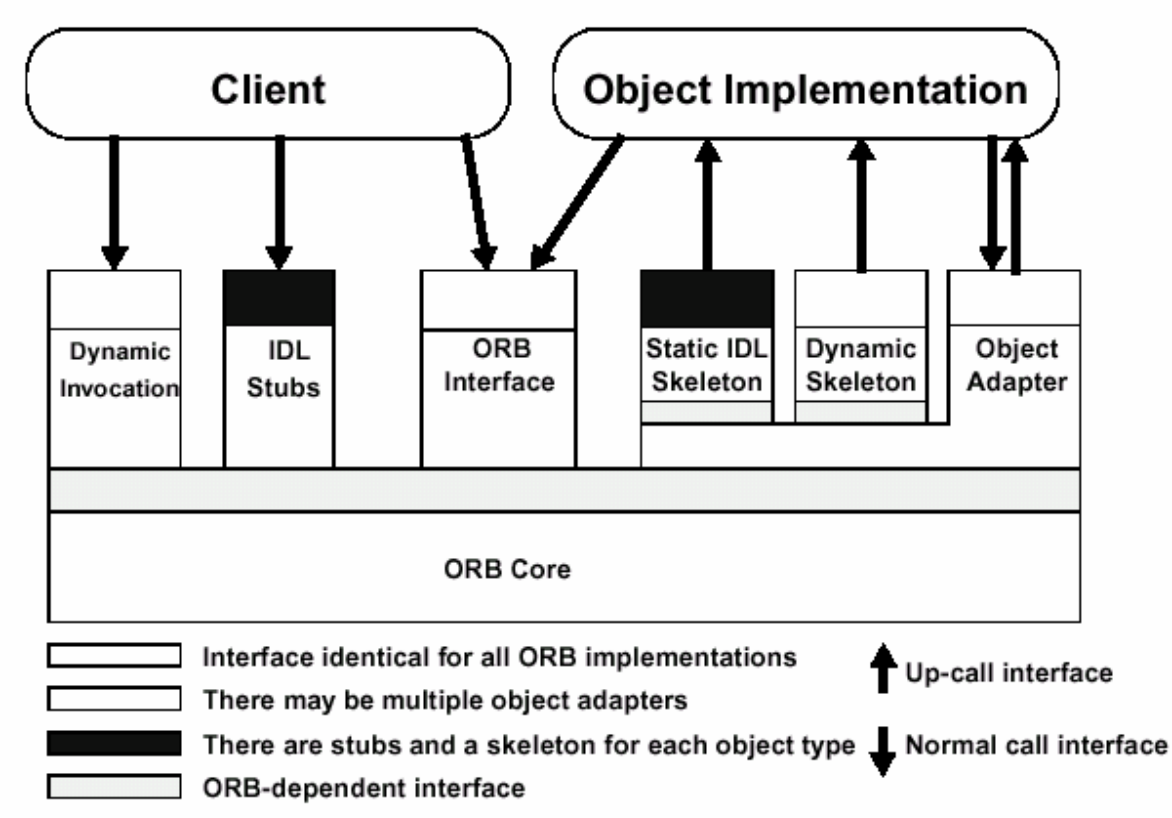
a) Classe.java – É a interface mapeada para a linguagem do ORB;
b) ClasseHolder.java – Proporciona suporte para parâmetros *out* e *inout*;
c) ClasseHelper.java – Contém vários métodos estáticos que dão suporte a classe principal, inclusive o método *narrow()* que tenta identificar um objeto CORBA como do tipo da sua classe;
d) ClasseStub.java – Contém a interface *stub* para a comunicação entre o objeto cliente e o ORB;
e) ClasseOperations.java – Contém interfaces e métodos declarados no arquivo "Classe.idl", que são utilizados no lado do servidor;
f) ClassePOA.java – Contém a interface *skeleton* para a comunicação entre o servidor, o *Portable Object Adapter* (POA) e o ORB; e
g) ClassePOATie.java – Contém interfaces e métodos que são utilizados pelo POA.

Segundo Brose (2001), estes arquivos contém interfaces e classes que implementam *Stubs*, *Skeletons* e outros códigos utilizados para suportar a aplicação distribuída CORBA. Com isto, resta desenvolver o objeto Cliente e o objeto Servidor (implementação do objeto) para o conteúdo da definição IDL poder executar em um ambiente distribuído.

3.3 ESTRUTURA DO ORB

Como mencionado anteriormente, a IDL proporciona as bases das conformidades sobre de que maneira pode ser requisitado um objeto através de um produto ORB. Porém, existem outras formas de requisitar um objeto. Para isso é necessário conhecer a estrutura do ORB. A Figura 4 ilustra como é a estrutura interna de um ORB.

Figura 4: Estrutura do ORB



Fonte: OMG (2001a)

Capeletto (1999) afirma que os *stubs* e os *skeletons* são responsáveis pelo "empacotamento" dos dados. Os *stubs* estão localizados entre o ORB e o cliente e gerenciam

as mensagens enviadas do cliente para o ORB. Os *skeletons* estão entre a implementação do objeto e o ORB e repassam as mensagens recebidas pelo ORB até a implementação do objeto.

3.3.1 STUB

O *stub* é gerado na compilação da definição IDL, portanto é estático. O *stub* contém interfaces e métodos que são responsáveis pelas requisições serem enviadas ao produto ORB de modo que seja transparente ao objeto cliente. Para um objeto cliente utilizar o *stub*, é necessário que o cliente possua uma referência para a implementação do objeto. Segundo Brose (2001), em linguagens orientadas a objeto os *stubs* são instanciados como objetos *proxy* locais que delegam as invocações.

A interface cliente-*stub* é padronizada pelo OMG para cada linguagem já homologada. Logo, o código gerado pode ser portado de um ORB para outro, desde que na mesma linguagem, ao contrário da interface *stub*-ORB que é proprietária, ou seja, cada fornecedor de ORB tem sua própria definição. Por isso, compiladores IDL e produtos ORB são sempre vendidos em conjunto.

Uma invocação através do *stub* é síncrona, salvo quando é utilizada a cláusula *oneway*. Chamadas síncronas bloqueiam o cliente, não retornando o controle até que seja devolvido um resultado ou uma exceção.

3.3.2 DYNAMIC INVOCATION INTERFACE (DII)

Permite a criação e invocação dinâmica de requisições a objetos CORBA, substituindo o uso do *stub*. Um cliente usa este tipo de interface para enviar uma invocação de qualquer operação para qualquer objeto disponível na rede (Siegel, 1996).

Conforme Capeletto (1999), os desenvolvedores não precisam fazer nada a mais para que seus objetos estejam preparados para invocações através da DII. Há quatro passos para efetuar uma requisição dinâmica:

a) identificar o objeto a ser invocado;
b) buscar sua interface;
c) construir a requisição;
d) efetuar a requisição e esperar o resultado.

3.3.3 SKELETON

O *skeleton* realiza a comunicação das requisições entre o ORB e a implementação do objeto. O *skeleton* recebe uma mensagem do ORB que uma operação da implementação do objeto deve ser executada. Segundo Brose (2001) o *skeleton* tem a responsabilidade de chamar o método correto na implementação de objeto correto.

Os *skeletons* conectam-se ao Adaptador de Objetos (descrito no tópico 3.3.5) através de uma interface proprietária de cada fornecedor. As requisições são efetuadas aos *skeletons* que por sua vez encaminha ao Adaptador de Objetos onde é executada a operação na implementação do objeto.

3.3.4 DYNAMIC SKELETON INTERFACE (DSI)

Os objetos podem ter seu comportamento alterado em tempo de execução. A DSI permite aos servidores despachar operações que não foram definidas em tempo de compilação (através do *skeleton*).

Segundo OMG (2001b), requisições totalmente dinâmicas através da DII e DSI promovem os mesmos resultados do que se fossem executadas estaticamente.

3.3.5 OBJECT ADAPTER

Conforme Brose (2001), são os Adaptadores de Objetos que conectam as implementações dos objetos com o produto ORB, e este ORB utiliza o Adaptador de Objetos para gerenciar o ambiente de execução das implementações de objetos. O Adaptador de Objeto tem como propósito gerar e interpretar referências de objetos e ativar e desativar as implementações dos objetos.

Atualmente, o padrão CORBA define uma única interface, o *Portable Object Adapter* (POA). Anterior ao POA, o *Basic Object Adapter* (BOA) era a interface padrão, que ainda é suportada por vários produtos ORB devido às aplicações mais antigas que utilizam o BOA. O POA foi adotado a partir da versão 2.2 do padrão CORBA e é a evolução do BOA.

3.3.6 ORB INTERFACE

A interface ORB contém funções do ORB que não dependem de qual adaptador de objeto está sendo usado. Essas operações são sempre as mesmas para todos os produtos ORB e todas as implementações de objetos, e podem ser executadas por clientes ou implementações de objetos. Algumas dessas operações parecem estar no ORB, outras parecem estar nas referências de objetos. Já que as operações ditas aqui são implementadas pelo próprio ORB, elas não são, de fato, operações nos objetos.

Existem várias operações pertencentes à interface ORB, mas destacam-se as operações de inicialização do produto ORB, de conversões entre objeto e string para identificação dos objetos, busca de interfaces, entre outras.

3.3.7 INTERFACE REPOSITORY (IR)

Segundo Siegel (1996), o repositório de interfaces é crucial para a operação do CORBA. O CORBA requer que cada produto ORB implemente um repositório de interfaces, permitindo que se conheça como as definições IDL são gravadas, modificadas e recuperadas.

Com o repositório de interfaces pode-se utilizar a Invocação Dinâmica (DII), pois a interface do objeto alvo deve ser encontrada no Repositório de Interface (IR). Outro propósito do IR é a verificação das assinaturas das operações requisitadas pelos clientes.

Um produto ORB pode acessar diversos Repositórios de Interface, e um Repositório de Interface pode ser compartilhado por vários produtos ORB simultaneamente e em tempo de execução.

3.3.8 IMPLEMENTATION REPOSITORY

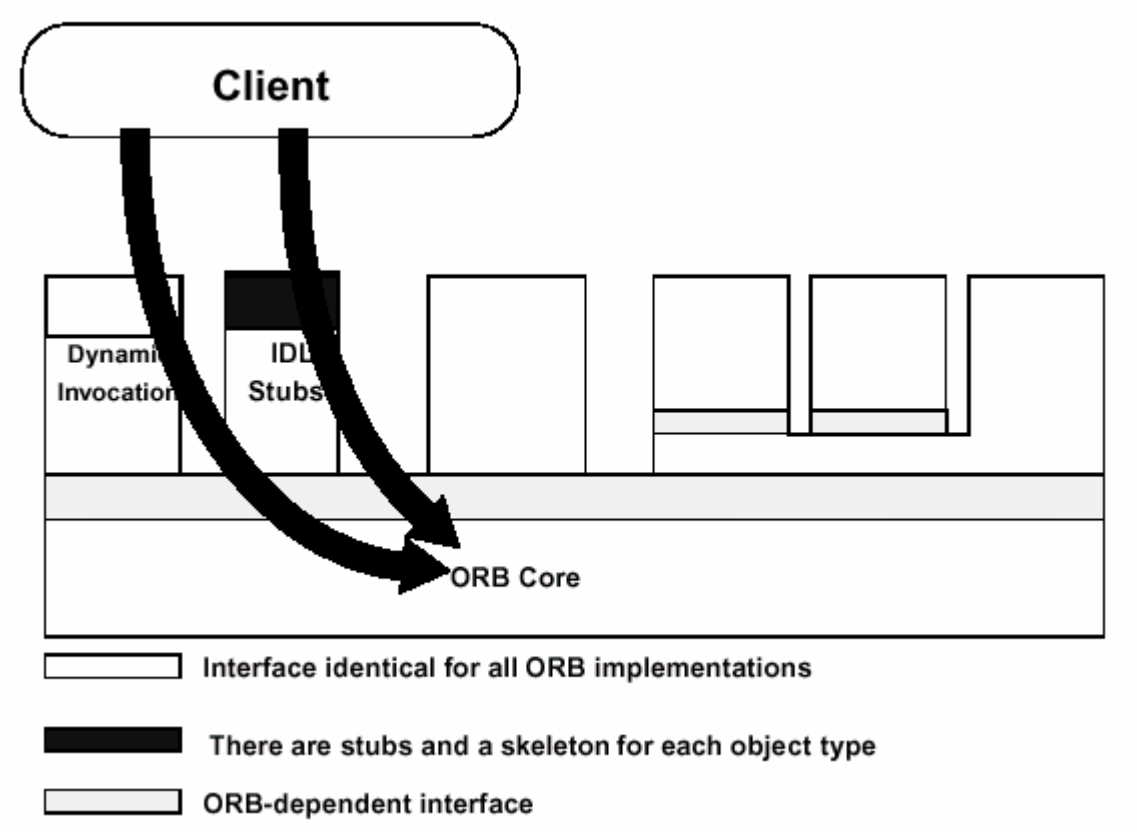
O Repositório de Implementações contém informações que permite ao ORB localizar e ativar implementações de objetos. Segundo OMG (2001b), apesar de muitas implementações no Repositório de Implementações serem específicas ao produto ORB ou ambiente operacional, o Repositório de Implementações é um lugar para gravar informações ordinárias, como instalação de implementações e controles relativos à ativação e execução de implementações de objetos.

3.4 EXECUÇÃO DO CORBA

Para facilitar o entendimento do funcionamento e da estrutura dos componentes do ORB, são demonstradas as etapas que uma requisição passa entre o cliente e o objeto de implementação em uma aplicação distribuída, segundo OMG (2001b).

Para fazer uma requisição, o cliente pode utilizar a Invocação Dinâmica ou utilizar a referência do objeto CORBA e o *Stub* gerado pelo compilador IDL. A Figura 5 ilustra a requisição do cliente sendo enviada ao ORB.

Figura 5: Requisição no lado do Cliente

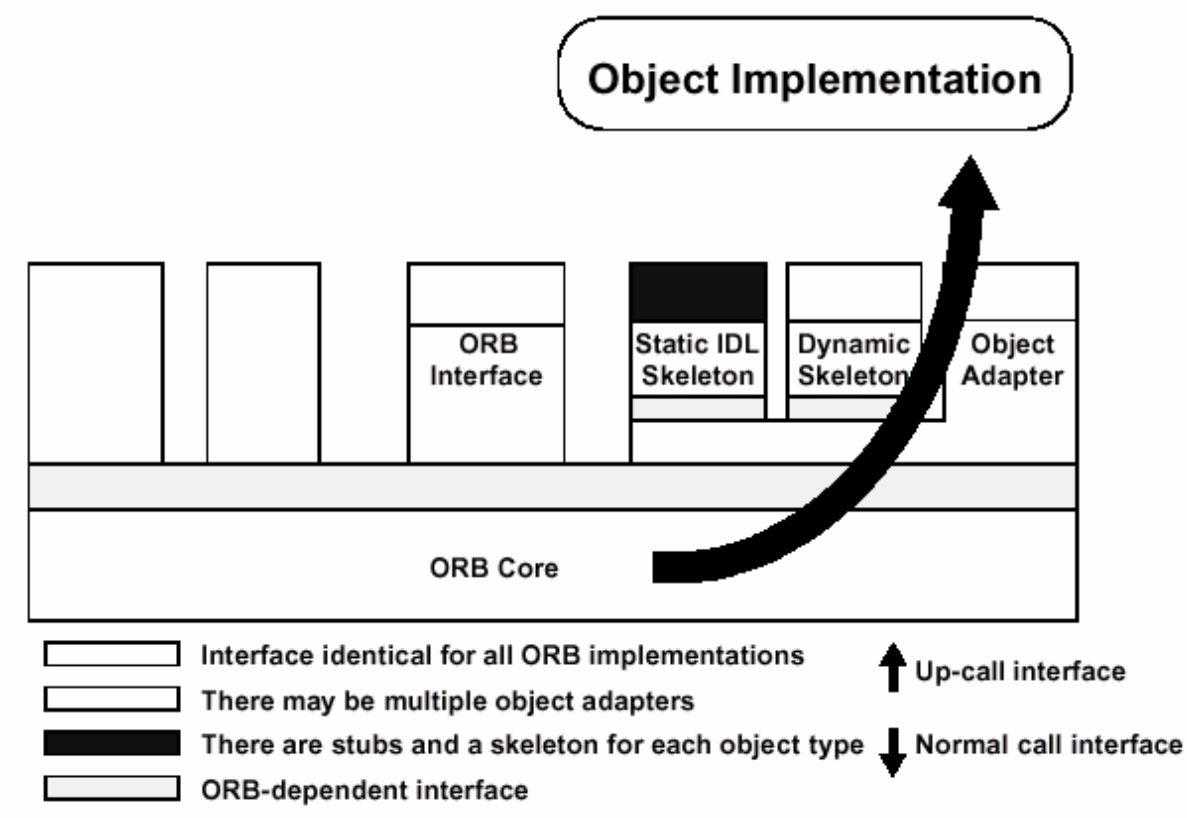


Fonte: OMG (2001a)

Esta requisição é feita ao produto ORB, que agora precisa resolver a localização deste objeto na rede e validar a requisição. Após localizar o objeto e validar a requisição, o ORB envia uma mensagem ao *skeleton* da implementação do objeto ou através da interface *skeleton* dinâmica (DSI), que por sua vez acionará o adaptador de objetos e fará a requisição à

implementação do objeto. A Figura 6 demonstra como a requisição é encaminhada ao correto objeto de implementação.

Figura 6: Requisição no lado do Servidor



Ao término da execução da operação na implementação do objeto, o caminho inverso é percorrido para retornar o resultado ao cliente.

Fonte: OMG (2001a)

3.5 VANTAGENS E DESVANTAGENS DO CORBA

Pelo fato do padrão CORBA constituir aplicações distribuídas, todas as vantagens e desvantagens citadas anteriormente sobre Sistemas Distribuídos se aplicam também ao CORBA.

Comparando o CORBA com seus principais concorrentes, verifica-se que o CORBA é totalmente independente, tanto de plataforma, como sistema operacional, protocolo de rede, linguagem de programação, e isto possibilita várias alternativas que nenhum produto similar

no mercado oferece. Outra grande vantagem é o código aberto, assim, é possível a customização de vários aspectos e um real entendimento por parte do desenvolvedor de software para a construção de suas aplicações.

Contudo, toda esta independência e abertura de possibilidades causam algumas desvantagens, como a velocidade de execução. De fato, muito código é executado para que seja possível a independência de plataforma, sistema operacional, protocolo de rede, linguagem de programação. Outro aspecto é a abrangência de produtos ORB, o que demanda tempo em um estudo para escolha do melhor produto ORB para uma determinada aplicação.

4 SERVIÇO DE PERSISTÊNCIA

A maioria absoluta dos sistemas de informação voltados à gestão empresarial depende fortemente de objetos persistentes, o que torna a persistência um fator crítico de sucesso para essa categoria de sistemas. Brose (2001) define que persistência é a capacidade de um objeto de sobreviver fora dos limites da aplicação que o criou. Normalmente isto significa que o objeto tem que ser gravado em um meio de armazenamento persistente.

Segundo OMG (1999) , o serviço de persistência apresenta informações persistentes como objetos de armazenamento em *datastores*. Um *datastore* é definido pelo OMG (1999) como uma entidade que administra dados, por exemplo um banco de dados, um sistema de arquivos, um esquema em um banco de dados relacional, etc.

Uma instância de um objeto pode ser ligada a um objeto de armazenamento no *datastore*, o que proporciona acesso direto ao estado deste objeto de armazenamento. Conforme OMG (1999), isto é chamado de **encarnação do objeto de armazenamento**.

4.1 PERSISTÊNCIA DE OBJETOS

Devido aos conceitos-chave da orientação a objeto, como a abstração e o encapsulamento do objeto, desde o início pode-se observar uma grande preocupação com a persistência dos objetos entre as várias invocações dos programas. A partir daí, várias formas e estratégias foram desenvolvidas por diferentes empresas e pessoas para tentar solucionar o “problema” da persistência dos objetos. Estas formas e estruturas são descritas a seguir.

4.1.1 NÍVEIS DE PERSISTÊNCIA

O nível de persistência pode ser interpretado como o escopo dos objetos. Segundo Brose (2001), os objetos podem ser transientes ou persistentes.

Os objetos transientes são aqueles criados em memória durante a execução do programa e mantém seu estado até que o programa que o criou seja encerrado ou o objeto é destruído.

Os objetos persistentes são os objetos armazenados que podem ser recuperados após o encerramento da execução do programa que os criou. Os objetos persistentes são recarregados para a memória para serem utilizados.

4.1.2 FORMAS DE PERSISTÊNCIA

Para obter persistência do estado dos objetos de um programa, as linguagens de programação e ambientes contam com três formas básicas de armazenamento:

a) **salvamento da imagem de uma sessão**: utilizada nos ambientes de programação Smalltalk e Lisp, salva todo o ambiente de uma sessão, salvando os estados de todos os objetos, variáveis e parâmetros de ambiente em um arquivo executável do sistema operacional. Tem como principal vantagem o armazenamento de qualquer tipo de objeto. Mas, é ineficiente e possui capacidade limitada para o compartilhamento de objetos. As desvantagens aumentam a medida que é maior a quantidade de objetos a serem armazenados (Khoshafian, 1994);

b) **arquivos de sistema operacional**: a estratégia mais empregada para armazenar objetos é utilizar arquivos do sistema operacional no armazenamento de textos ou registros estruturados que não contêm endereços de memória (ponteiros). A principal vantagem é a eficiência para grandes quantidades de objetos persistentes. Uma desvantagem desta forma é a persistência de objetos complexos porque não existem técnicas eficientes e eficazes para armazenar os objetos em arquivos. (Khoshafian, 1994);

c) **persistência através da rede**: com os objetos distribuídos em redes de computadores é possível que o programa, antes de encerrar a sua execução, transporte os seus objetos em memória para um outro nodo qualquer da rede. Quando o programa é executado novamente, os objetos que foram criados por ele na execução anterior podem ser recuperados nesta execução através da rede. A principal vantagem é que os objetos estão sempre em memória. Como desvantagens, a dificuldade na implementação e a possibilidade de não haver um nodo na rede capaz de receber os objetos.

Ambler (1998) cita 4 abordagens para a persistência de objetos em arquivos de sistema operacional. São eles, em arquivos simples, em bancos de dados relacionais, em bancos de dados orientados a objeto e banco de dados híbridos ou objeto-relacionais.

4.1.2.1 PERSISTÊNCIA EM ARQUIVOS

Uma abordagem comum para tornar os objetos persistentes é salvá-los em arquivos simples. Um arquivo simples é um arquivo sendo usado para armazenar informações dos objetos (Ambler, 1998).

Dependendo da linguagem utilizada, é possível variar as formas de armazenamento em arquivos, como por exemplo, armazenar os objetos utilizando a seqüência lização de objetos (Java) ou armazenar os atributos dos objetos (qualquer linguagem).

Na segunda variação muitos aspectos são importantes, onde se destacam a utilização de um dicionário de dados, forma de acesso dos arquivos, quantidade de objetos por linha, as posições dos atributos dentro do arquivo, entre outros.

4.1.2.2 PERSISTÊNCIA EM BANCO DE DADOS RELACIONAL

Segundo Ambler (1998), os bancos de dados relacionais são utilizados pela maioria dos desenvolvedores de aplicações orientadas a objeto para armazenar os objetos persistentes. Algumas razões para utilizar banco de dados relacional são:

a) as empresas possuem um grande investimento em tecnologia relacional;
b) os dados legados da empresa são bens que não podem ser perdidos ou extraviados;
c) as aplicações corporativas desenvolvidas são muito transacionais, uma característica intrínseca de bancos de dados relacionais;
d) banco de dados relacional é uma tecnologia já utilizada e confiável;
e) outras aplicações que não são orientadas a objeto podem precisar acessar informações de uma aplicação orientada a objeto.

Os objetos formam uma unidade que encapsula atributos e operações (Ambler, 1998). Os bancos de dados relacionais são eficientes para representar os atributos, mas são muito limitados quanto às operações. Pode-se tentar estabelecer uma analogia direta entre objetos e tabelas, e entre atributos e colunas. Entretanto, alguns aspectos merecem destaque nesta

analogia, para garantir que todas as características do modelo de objetos sejam reproduzidas com precisão pelo banco de dados relacional. Abaixo, está um roteiro resumido que Ambler (1998) sugere para a derivação do modelo de objetos em tabelas relacionais:

a) decidir quais classes e atributos devem ser persistentes;
b) criar uma tabela para cada classe. É conveniente adotar como nome da tabela o mesmo nome da classe para aumentar a legibilidade do software;
c) cada atributo (primitivo) será uma coluna na tabela. Se o atributo for complexo (composto por vários tipos do SGBD), ou adiciona-se uma nova tabela para o atributo ou separa-se o atributo em várias colunas (primitivas) na tabela da classe;
d) a coluna da chave primária será o único identificador de instância. O identificador deve ser transparente para o usuário, para que mudanças na chave por razões administrativas não causem impacto em toda a aplicação;
e) para associações, o identificador deve ser inserido como um atributo na tabela da classe que participa com multiplicidade 1;
f) para associações muitos-para-muitos, deve-se criar uma tabela associativa, que englobe os identificadores das classes envolvidas;
g) para agregações, o identificador do todo deve fazer parte do identificador da parte na tabela criada (chave primária composta); e
h) para herança há duas estratégias distintas: (1) se não há necessidade de gerenciar os objetos descendentes de forma genérica, os atributos herdados são copiados para todas as tabelas que representam classes descendentes. Nenhuma tabela representará a classe abstrata; (2) se há a necessidade de gerenciar de forma genérica, a classe abstrata é uma tabela própria, para a qual as descendentes fazem referência.

Cada instância da classe é representada por uma linha na tabela. As diferenças entre a abordagem relacional e o paradigma dos objetos são “camufladas” desta maneira nesta abordagem para obter as principais vantagens de cada conceito.

4.1.2.3 PERSISTÊNCIA EM BANCO DE DADOS ORIENTADO A OBJETO

Uma abordagem para incorporar capacidades de orientação a objeto em bancos de dados é desenvolver uma linguagem para banco de dados e sistemas gerenciadores de banco de dados (SGBD) inteiramente novos e com características de orientação a objeto.

Segundo Ambler (1998), um sistema de gerenciamento de banco de dados orientado a objetos (SGBDOO) é utilizado para armazenar tanto os atributos como os métodos dos objetos. Isto é diferente dos bancos de dados tradicionais, que só armazenam dados.

Os seguintes aspectos devem ser observados em um banco de dados orientado a objeto:

a) suporte a objetos complexos;
b) identidade de objetos;
c) encapsulamento de objetos;
d) suporte a tipos ou classes;
e) herança entre classes ou tipos;
f) funções básicas de persistência (criar, ler, atualizar e excluir);
g) gerenciamento de grandes bancos de dados;
h) suporte a vários usuários;
i) segurança; e
j) recuperação de falhas de hardware ou software.

4.1.2.4 PERSISTÊNCIA EM BANCO DE DADOS HÍBRIDO OU OBJETO-RELACIONAL

Um banco de dados híbrido é um mecanismo de persistência que reúne as características de banco de dados relacional e de banco de dados orientado a objeto.

Em um banco de dados híbrido é possível armazenar tabelas com registros e colunas, e, ao mesmo tempo, armazenar objetos. Segundo Ambler (1998), isto pode causar confusão entre os desenvolvedores de aplicações, por exemplo, em uma aplicação utilizar tabelas e objetos para armazenar as informações. O desenvolvimento e a manutenção desta aplicação pode ser assim dificultado.

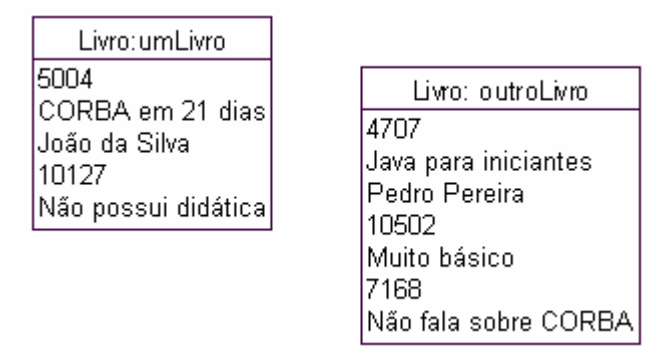
4.1.3 ESTRATÉGIAS DE PERSISTÊNCIA

Há muitas estratégias alternativas para o armazenamento de objetos. Segundo Khoshafian (1994), o principal objetivo das estratégias de armazenamento de objetos é proporcionar o armazenamento e a recuperação dos objetos de forma eficiente. Khoshafian (1994) cita três possíveis estratégias para o armazenamento de objetos.

Os exemplos desta seção, mostrados a seguir, partem da definição de que um livro tem identificador, título e autor; e pode possuir nenhuma resenha ou várias resenhas. Uma resenha tem os atributos: identificador da resenha, identificador do livro e descrição da resenha. Uma resenha sempre pertence a um livro.

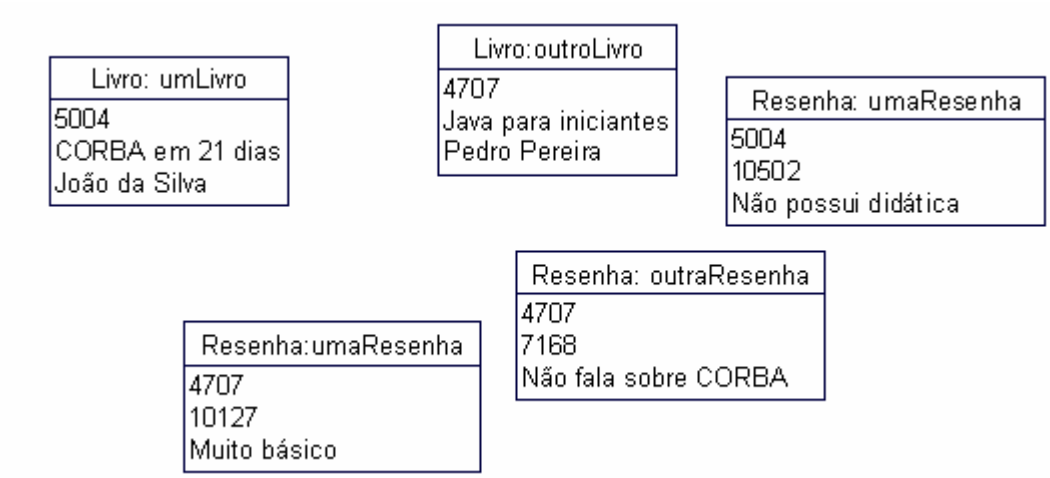
O **modelo de armazenamento direto** consiste em armazenar objetos complexos o mais "diretamente" possível, refletindo o modo como os objetos são definidos ou ocupados no *datastore*. Assim, os subobjetos são armazenados perto de seus superobjetos sempre que possível. A principal vantagem desta estratégia é que a recuperação do objeto é muito eficiente, pois a tendência do objeto é não ser decomposto. A Figura 7 ilustra o modelo de armazenamento direto.

Figura 7: Exemplo do modelo de armazenamento direto



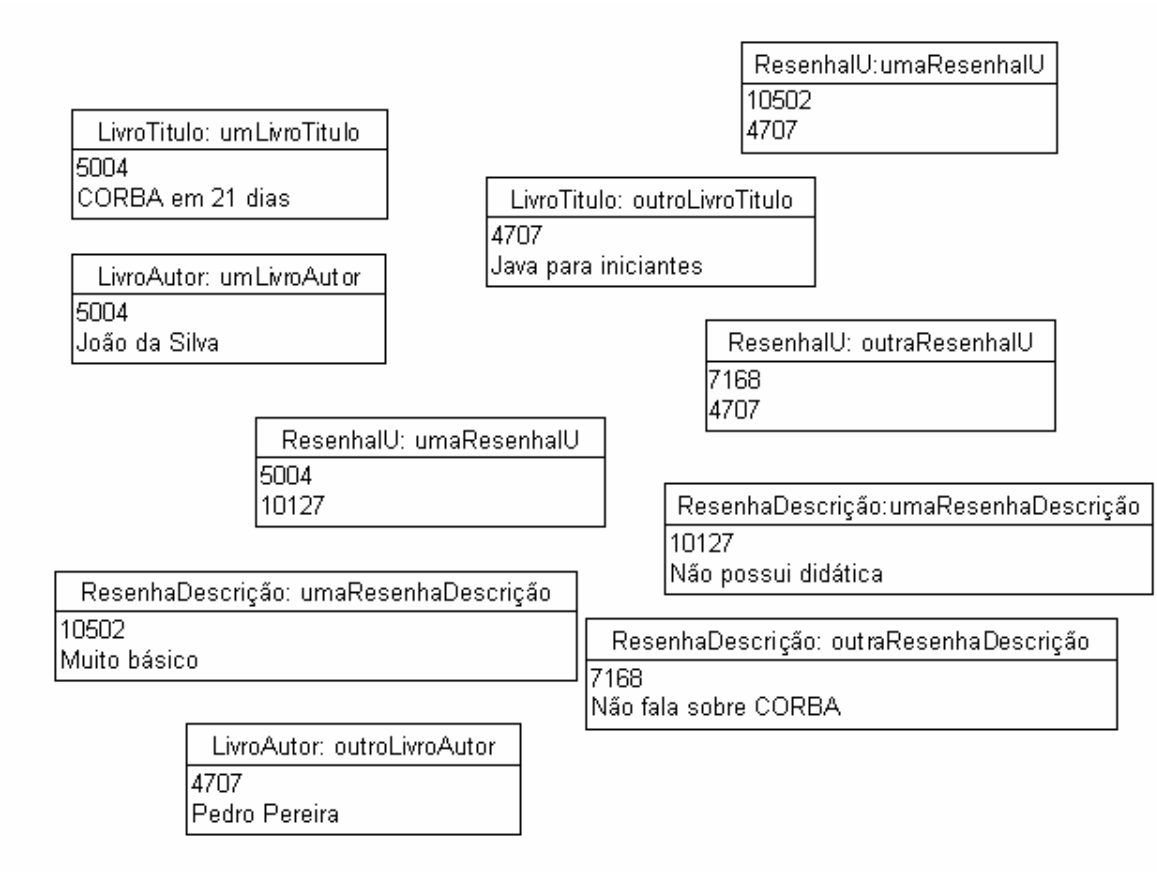
O **modelo de armazenamento normalizado** é parecido com o modelo de armazenamento direto do objeto em um *datastore*, porém, aplicando-se as regras de normalização de sistemas. Informalmente, isso significa o armazenamento de um conjunto de objetos homogêneos para cada tipo de objeto. A principal vantagem desta estratégia é a diminuição da redundância das informações. A Figura 8 demonstra o modelo de armazenamento normalizado.

Figura 8: Exemplo do modelo de armazenamento normalizado



O **modelo de armazenamento decomposto** consiste no emparelhamento de cada valor de atributo de um objeto com o substituto dele. Por exemplo, um objeto com três atributos é decomposto em três objetos onde, cada objeto possui um identificador do objeto e um atributo do objeto. As principais vantagens desta estrutura são a simplicidade, a generalidade e a performance mais uniforme para diferentes tipos de acesso. A principal desvantagem está na complexidade de reconstruir o objeto. A Figura 9 ilustra o modelo de armazenamento decomposto.

Figura 9: Exemplo do modelo de armazenamento decomposto



4.2 SERVIÇO DE PERSISTÊNCIA SEGUNDO O OMG

Siegel (1996) afirma que a arquitetura OMA especifica regras sobre certos aspectos de objetos. Desta forma, os clientes sabem antecipadamente o que esperar de um objeto remoto na rede. Pode-se pensar nesta parte do OMA como um tipo de “etiqueta” de objeto; se todos os objetos obedecerem isto, interações ocorrerão suavemente e ninguém é surpreendido por qualquer coisa que possa acontecer. Talvez a parte mais importante desta “etiqueta” dos objetos envolve o *Persistent State Service* (PSS).

A arquitetura OMA contém o PSS para unificar os processos de armazenamento e recuperação de seu estado persistente, proporcinando os seguintes padrões.

a) definir se o armazenamento é automático ou explicitamente controlado pelo cliente;
b) definir se controle está acima do estado de um único objeto ou de um esquema de objetos relacionados;

c) padronizar as interfaces referentes a persistência, indiferentemente dos vários métodos de armazenamento que podem ser utilizados; e
d) coibir através da granularidade, a extensiva gama de operações de armazenamento e entidades armazenadas.

Em vista a todas estas alternativas, o serviço de persistência tenta ainda proporcionar um objeto de serviço persistente orientado a objeto com interfaces IDL padronizadas. A principal vantagem disto é que aplicações diferentes podem se comunicar mais facilmente.

Conforme Brose (2001), pelo fato do serviço de persistência somente proporcionar o armazenamento persistente de objetos, ele não proporciona uma interface completa como um Sistema Gerenciador de Banco de Dados Orientado a Objeto (SGBDOO). São necessários outros serviços como consultas e transações. A omissão dessas características na especificação do serviço de persistência é simples: a separação de conceitos. O OMG sempre tem separado as especificações dos serviços de persistência, transação e consulta, mas estão intimamente interligados.

Segundo OMG (1999), ao implementar uma aplicação, utilizando o serviço de persistência, o desenvolvedor é responsável por especificar os tipos de objetos de armazenamento necessários. O serviço de persistência proporciona dois modos para definir o esquema do *datastore* e da aplicação:

a) utilizando a *Persistent State Definition Language* (PSDL); ou
b) diretamente em uma linguagem de programação, denominada Persistência Transparente.

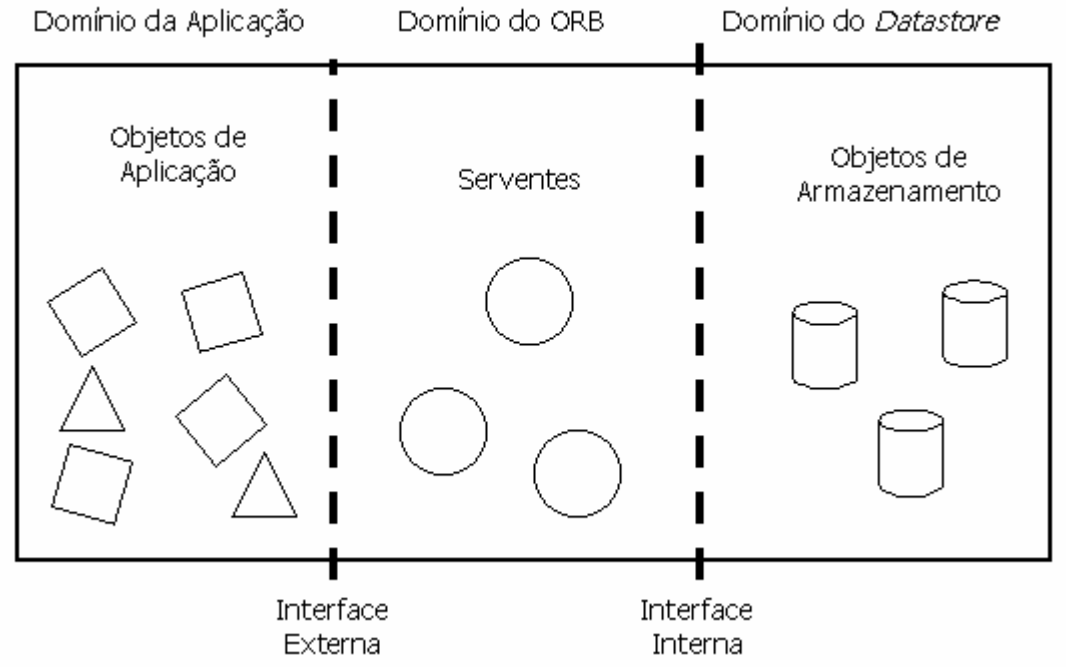
PSDL é um superconjunto da IDL, com 5 novos construtores: *storagetype*, *storagehome*, *catalogue*, *storagetype abstract* e *storagehome abstract*. A PSDL é explanada na seção 4.2.2.

Uma implementação do serviço de persistência que apóia persistência transparente lhe permite especificar o armazenamento dos objetos diretamente em uma linguagem de programação. Este conceito é aprofundado na seção 4.2.3.

4.2.1 MODELO DE ADMINISTRAÇÃO DE DADOS

A especificação do serviço de persistência do OMG prevê o Modelo de Administração de Dados (*Datastore Model*) com o objetivo de padronizar as interfaces internas utilizadas no serviço de persistência dos objetos. O serviço de persistência consiste na implementação dos objetos de armazenamento e os serventes. A Figura 10 demonstra como os objetos são dispostos no contexto do serviço de persistência.

Figura 10: Disposição dos objetos no serviço de persistência



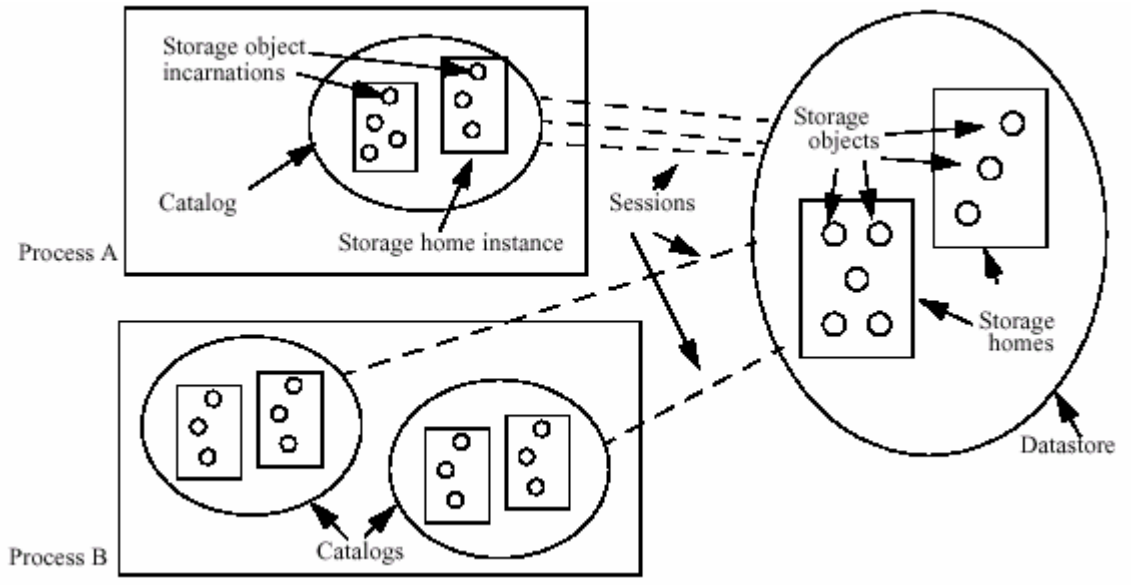
Fonte: adaptado de OMG (1999)

Os objetos de aplicação enviam as mensagens aos serventes, que por sua vez executam as operações correspondentes nos objetos de armazenamento. Em geral, um servente é implementado para cada tipo de objeto de armazenamento, mas isto não é definido como regra pelo OMG.

4.2.1.1 ESTRUTURA DE UM DATASTORE

Analisando os objetos de armazenamento em particular, o OMG define vários conceitos referentes à estrutura de um *datastore* para melhorar a padronização e facilitar o desenvolvimento das aplicações. A Figura 11 ilustra a estrutura de um *datastore*.

Figura 11: Estrutura de um datastore



Fonte: OMG (1999)

Conceitualmente, um *datastore* é um conjunto de casas de armazenamento (*storage homes*). Cada casa de armazenamento tem um tipo. Segundo OMG (1999), dentro de um *datastore*, uma casa de armazenamento é singular: existe no máximo uma casa de armazenamento de um determinado tipo em cada *datastore*.

Uma casa de armazenamento contém objetos de armazenamento. Um tipo de casa de armazenamento pode derivar de outro tipo de casa de armazenamento. Cada objeto de armazenamento (*storage object*) tem um identificador único em sua casa de armazenamento (*short-pid*) e um identificador único global (*pid*). A extensão do *pid* é todos os objetos de armazenamento que podem ser acessados pelo mesmo catálogo. Um catálogo é a instância do *datastore* através de sessões (OMG, 1999).

Cada objeto de armazenamento tem um tipo que define o seu estado e operações. Um tipo de objeto de armazenamento pode derivar de outro tipo de objeto de armazenamento. Uma casa de armazenamento pode conter somente objetos de armazenamento de um determinado tipo.

Os objetos de armazenamento podem ser encarnados (instanciados) por um objeto de aplicação. As casas de armazenamento também podem ser instanciadas. Um objeto encarnado

é gerenciado pela instância da casa de armazenamento a que pertence, e a instância da casa de armazenamento por sua vez é gerenciada pelo catálogo.

4.2.2 PERSISTÊNCIA ATRAVÉS DA PSDL

Objetos de armazenamento e casas de armazenamento podem ser definidos através da PSDL. Segundo a OMG, a PSDL é uma linguagem declarativa que obedece as mesmas regras da IDL, porém com a sua gramática estendida.

Uma especificação PSDL pode conter qualquer construtor IDL. Um arquivo fonte PSDL tem que ter a extensão ".psdl". A descrição da gramática de PSDL usa a mesma notação que a especificação CORBA. O Quadro 3 exemplifica uma definição PSDL.

Quadro 3: Exemplo de definição PSDL

```
// arquivo Pessoa.psdl

  abstract storagetype Pessoa {
    readonly state long iu_pessoa;
    state string nm_pessoa;
    state string nr_telefone;
  };

  abstract storagehome PessoaTab of Pessoa {
    Pessoa criar(in long iu_pessoa,
                 in string nm_pessoa,
                 in string nr_telefone);
  };

  catalog DeptoPessoal {
    provides PessoaTab pessoas;
  };
```

Uma ferramenta que implementa o serviço de persistência processará este arquivo e gerará o código em sua linguagem de programação designada. Por exemplo, se o objetivo é utilizar a linguagem de programação Java, a ferramenta deverá gerar uma interface de Java para cada *abstract storagetype*, *abstract storagehome* e *catalog*.

Realizando uma analogia entre uma definição PSDL e um Banco de Dados Relacional, o catálogo será a conexão com o banco de dados, a casa de armazenamento será equivalente a uma tabela de dados e o objeto de armazenamento corresponderá a um registro na tabela.

Segundo OMG (1999), quando um arquivo-fonte PSDL é compilado, implicitamente, as definições contidas no arquivo "*CosPersistentState*.psdl" são importadas para o arquivo que está sendo compilado. O PSDL interage com os tipos abstratos *storagetypes* e *storagehomes* que estão definidos no módulo *CosPersistentState*. O Anexo 1 lista a definição PSDL do módulo *CosPersistentState*. Esta listagem encontra-se na especificação do serviço de persistência (OMG, 1999).

4.2.3 PERSISTÊNCIA TRANSPARENTE

Uma implementação do serviço de persistência que apóia a definição de objetos de armazenamento diretamente em uma linguagem de programação é dita que apóia a persistência transparente. Conforme OMG (1999), com esta capacidade, não há nenhuma necessidade para uma especificação PSDL em separado do esquema. O mecanismo de persistência transparente também tenta permitir que qualquer classe da linguagem possa ser persistente, embora existam restrições: a encarnação do objeto tem que estar completa para o programa utilizar o objeto; e o serviço deve ser capaz de identificar os objetos que foram alterados ou necessitam ser confirmados no *datastore*.

O benefício mais visível é que os atributos dos objetos podem ser representados diretamente em campos ao invés de necessitar métodos acessores e modificadores que fazem chamadas para a implementação do serviço de persistência.

Na persistência transparente é preciso ter certeza que a encarnação de um objeto está carregada antes do programa tentar acessar este objeto. Deve também ser capaz de determinar quais objetos mudaram e quais precisam ser confirmados (OMG, 1999).

Uma vez que a definição de esquemas e a manipulação de dados são realizadas diretamente na linguagem de programação selecionada, como Java ou C++, a maioria da descrição da persistência transparente está, em termos, na linguagem utilizada para a implementação do serviço de persistência.

Com a persistência transparente, porém, as casas de armazenamento não devem ser definidas. Segundo OMG (1999), as casas de armazenamento devem ser omitidas e nenhuma operação deve estar implicitamente definida.

4.2.4 CONFORMIDADES REQUERIDAS PELO OMG

Segundo OMG (1999), uma implementação compatível tem que implementar o módulo *CosPersistentState* completamente em pelo menos uma linguagem de programação para a qual a especificação define um mapeamento. Também tem que proporcionar uma ferramenta que lê especificações de PSDL e gera código nesta linguagem de programação.

Há duas características opcionais: apoio de transação e persistência transparente.

Uma implementação compatível que apóia transações conforme é especificado no serviço de persistência pode reivindicar que seja uma implementação de Serviço de Estado Persistente compatível com apoio de transação.

Uma implementação compatível que apóia persistência transparente pode reivindicar ser uma implementação de Serviço de Estado Persistente compatível com persistência transparente.

5 DESENVOLVIMENTO DO TRABALHO

Para o projeto e implementação do serviço de persistência, observou-se a necessidade de desenvolver um protótipo de aplicação distribuída que utilizasse o serviço de persistência desenvolvido para demonstrar e validar este serviço.

A implementação ORB selecionada para ser utilizada pela implementação do trabalho é o OpenORB 1.2.0, da Exolab. A Exolab é uma organização informal que trabalha no desenvolvimento de projetos de software com código-fonte aberto (Daniel, 2001). Os principais motivos que levaram à escolha deste produto são a versão atualizada da especificação CORBA, o seu desenvolvimento em Java, o código-fonte aberto do ORB e dos serviços implementados, e a quantidade de serviços implementados. Segundo Daniel (2001), este ORB contém as seguintes características:

a) completamente compatível com CORBA 2.4.2;
b) completamente desenvolvido em Java;
c) ORB completamente *multithreaded*;
d) compatível com o Java JDK 1.2 e 1.3;
e) ORB verdadeiramente modular;
f) configuração através de arquivos XML;
g) adaptadores de Objeto múltiplos: BOA e POA;
h) suporte a protocolos de comunicação IIOP 1.2 e Bi GIOP Direcional;
i) interceptadores portáteis;
j) serviço de nome embutido;
k) suporte a DynAny, DII e DSI;
l) suporte de conjuntos de códigos;
m) compilador de IDL reutilizável; e
n) várias ferramentas: IDL para HTML, IDL para RTF.

Conforme Daniel (2001), o OpenORB tem os seguintes serviços implementados à disposição dos desenvolvedores:

a) serviço de controle de concorrência;
b) serviço de nome (extensão);
c) serviço de eventos;
d) serviço de persistência;

e) serviço de propriedades;
f) serviço de tempo;
g) serviço de negociação;
h) serviço de transação; e
i) serviço de notificação.

Apesar do OpenORB já possuir todos esses serviços disponíveis, nenhum destes serviços será utilizado, inclusive o serviço de persistência. Os únicos serviços que serão utilizados pelo protótipo são o serviço de nomes embutido no OpenORB e o serviço de persistência desenvolvido neste trabalho.

A forma de persistência utilizada neste trabalho é em arquivos de sistema operacional, através do banco de dados relacional MS Access 2000. A estratégia empregada neste trabalho é o modelo de armazenamento normalizado. Os serventes são implementados na linguagem Java e acessam o banco de dados através de *Java DataBase Connectivity* (JDBC). Deitel (2001) afirma que JDBC realiza uma "ponte" entre o programa Java e a fonte de dados.

O serviço de persistência implementado é compatível com persistência transparente, de forma que o módulo *CosPersistentState* não é implementado, logo não oferece suporte a especificações PSDL.

O serviço de persistência deve ser capaz de criar, atualizar e excluir os objetos do banco de dados, e recuperá-los, conforme seja solicitado pelos clientes a qualquer momento durante a execução da aplicação ou em execuções posteriores.

Conforme descrito na seção 4.2.3, o serviço de persistência transparente é embutido no código da aplicação. Logo, para uma melhor compreensão do serviço de persistência desenvolvido o protótipo de aplicação é abordado primeiramente, para em seguida abordar o serviço de persistência implementado neste protótipo de aplicação.

5.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

Tanto o protótipo de aplicação quanto o serviço de persistência foram desenvolvidos utilizando as mesmas técnicas e ferramentas.

A ferramenta utilizada para a especificação do protótipo e do serviço de persistência é o Rational Rose 2000 e a técnica empregada é a UML. O Rational Rose é uma ferramenta da Rational para modelagem de sistemas orientados a objeto que utiliza os conceitos e a notação da UML (Rational, 2001). Atualmente, é uma das ferramentas com maior aceitação no mercado e contempla todos os conceitos da UML, pois os idealizadores da UML são também os criadores do Rational Rose. A UML é bem conceituada entre a comunidade de informática e representa os conceitos de orientação a objeto de uma forma consistente.

A implementação do protótipo e do serviço de persistência ocorreu através da ferramenta JBuilder 5, utilizando a linguagem de programação Java. O JBuilder é uma ferramenta da Borland implementada totalmente em Java, utilizada para o desenvolvimento de aplicativos ou *applets* Java (Armstrong, 1998). O desenvolvimento dos programas Java através do JBuilder é totalmente visual. A linguagem de programação Java suporta a maioria dos conceitos de orientação a objeto, além de ser uma linguagem robusta e independente de plataforma.

5.2 PROTÓTIPO DE APLICAÇÃO

O protótipo consiste em uma aplicação de controle de resenhas que tem como objetivo controlar as resenhas dos livros de uma biblioteca.

O servidor e os serventes serão implementados utilizando a linguagem de programação Java, conectando-se no banco de dados utilizando JDBC, e os clientes são implementados em aplicativos Java e *applets* Java.

Segundo Deitel (2001), um aplicativo Java é um programa que é executado utilizando o interpretador Java a partir do computador local do usuário. Já um *applet* Java é um programa de porte menor armazenado em um computador remoto que e é executado pela *World Wide Web* através de um navegador.

5.2.1 ESPECIFICAÇÃO

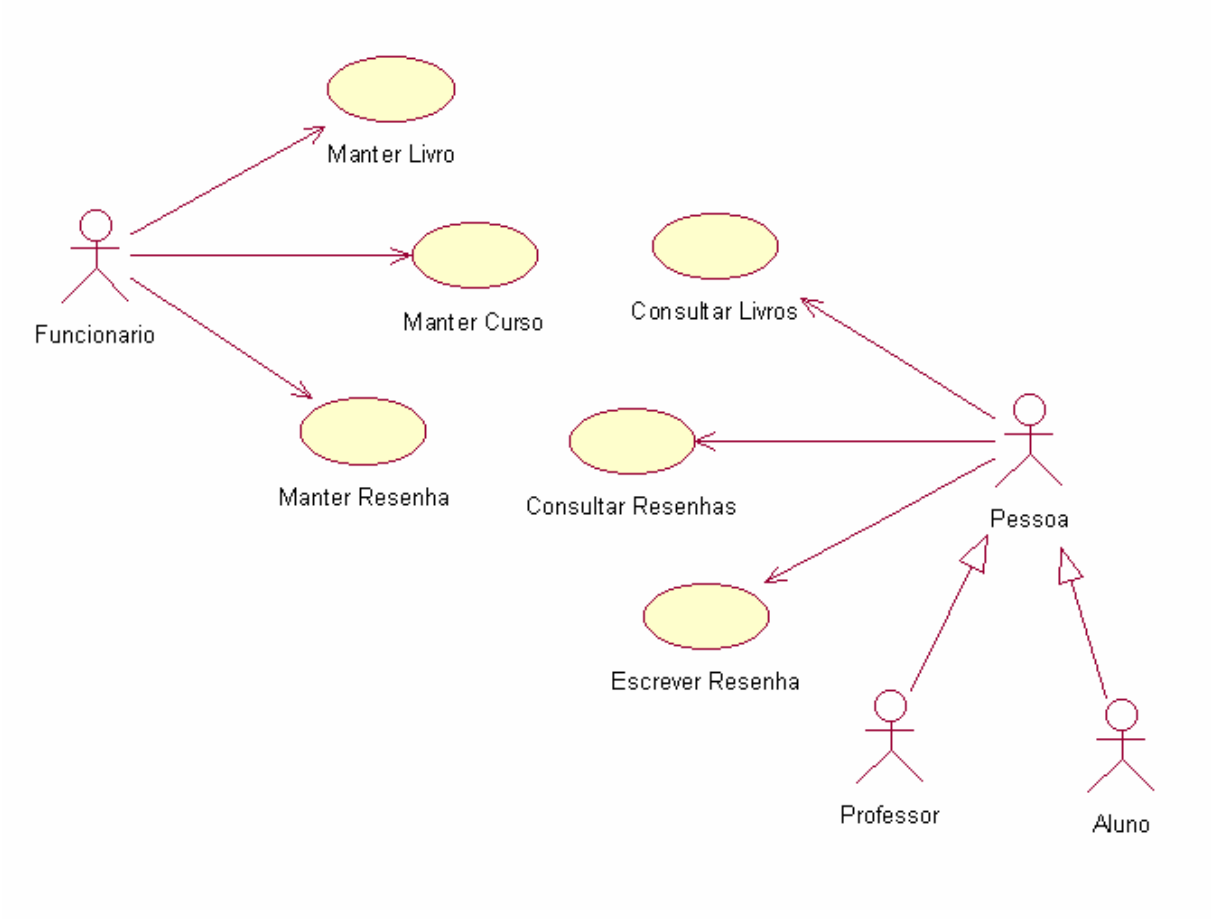
A especificação do protótipo de aplicação utiliza os conceitos da orientação a objeto e é baseada na UML. Os diagramas de casos de uso, classes e seqüência são utilizados para a especificação deste protótipo.

O protótipo consiste em manter os cursos, professores, alunos, livros e resenhas de uma escola qualquer. Os cursos, professores, alunos, livros e resenhas podem ser mantidos somente por funcionários da escola. Os professores e alunos podem somente consultar os livros e resenhas e escrever novas resenhas. A resenha pode ser escrita também através da Internet, utilizando um *applet* Java.

5.2.1.1 DIAGRAMA DE CASOS DE USO

Na aplicação de controle de resenhas, os casos de uso observados são representados pela Figura 12.

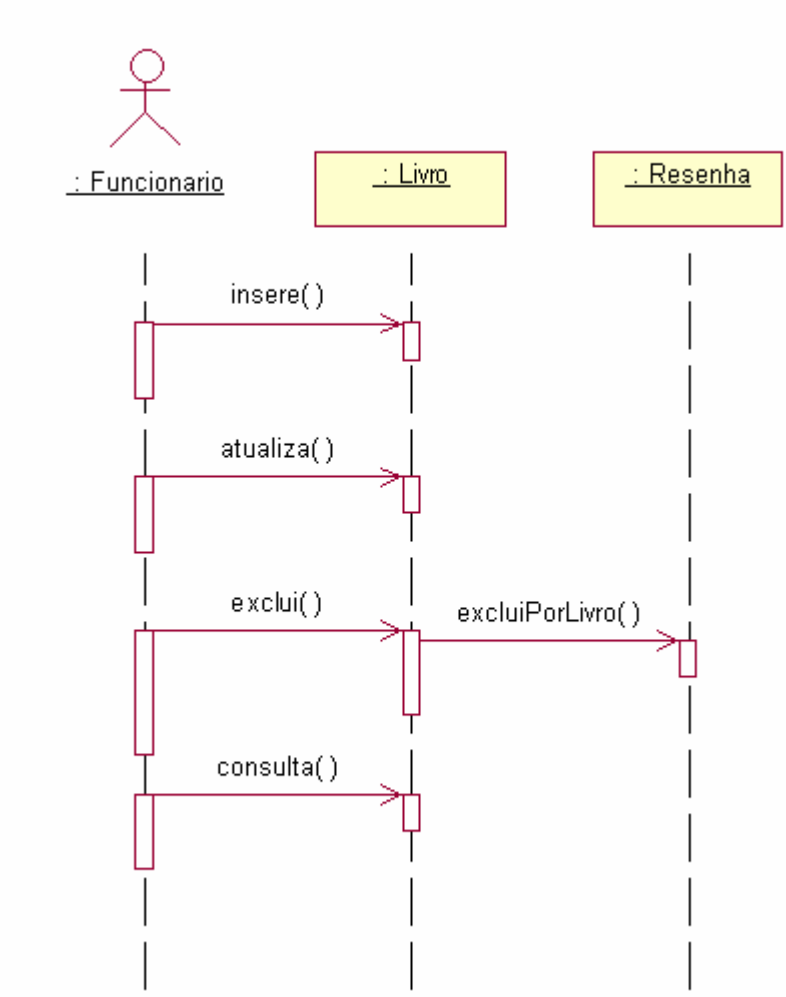
Figura 12: Diagrama de casos de uso (aplicação)



A seguir, são descritos os casos de uso principais:

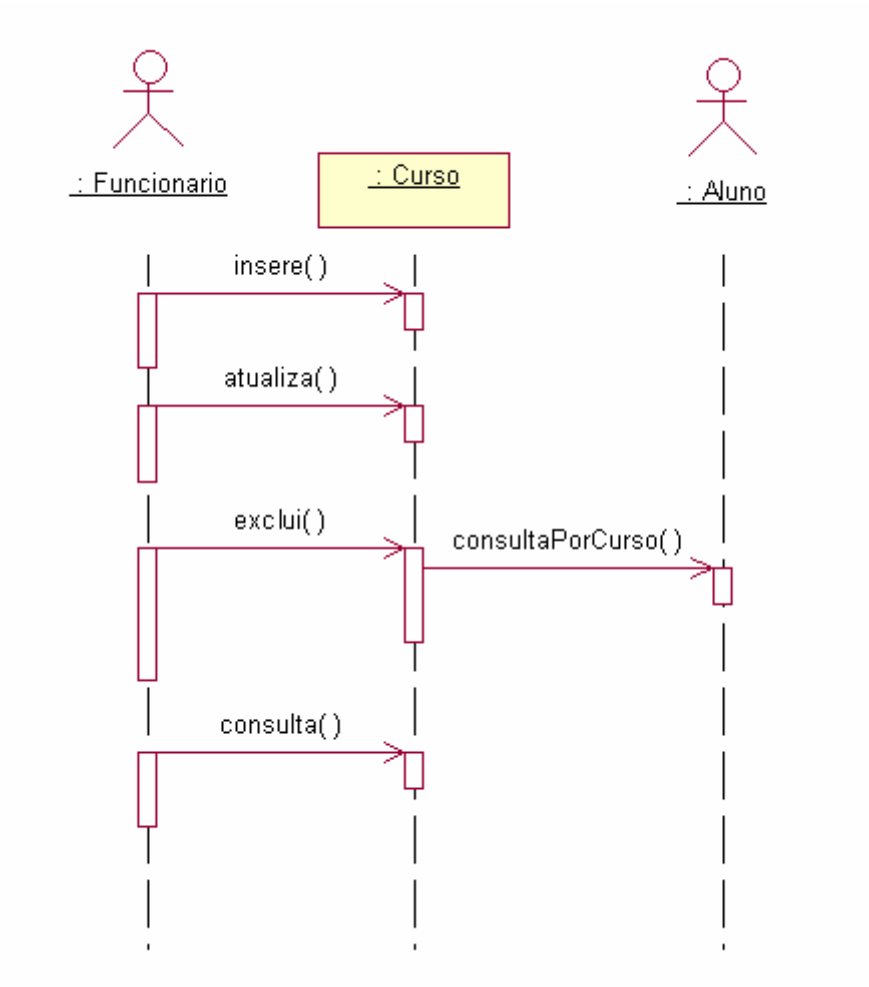
a) **Manter Livro**: Permite consultar, inserir, alterar e excluir livros. Ao excluir um livro, é necessário excluir também as resenhas referentes aquele livro. A Figura 13 representa o diagrama de seqüência deste caso de uso;

Figura 13: Diagrama de seqüência - Manter Livro



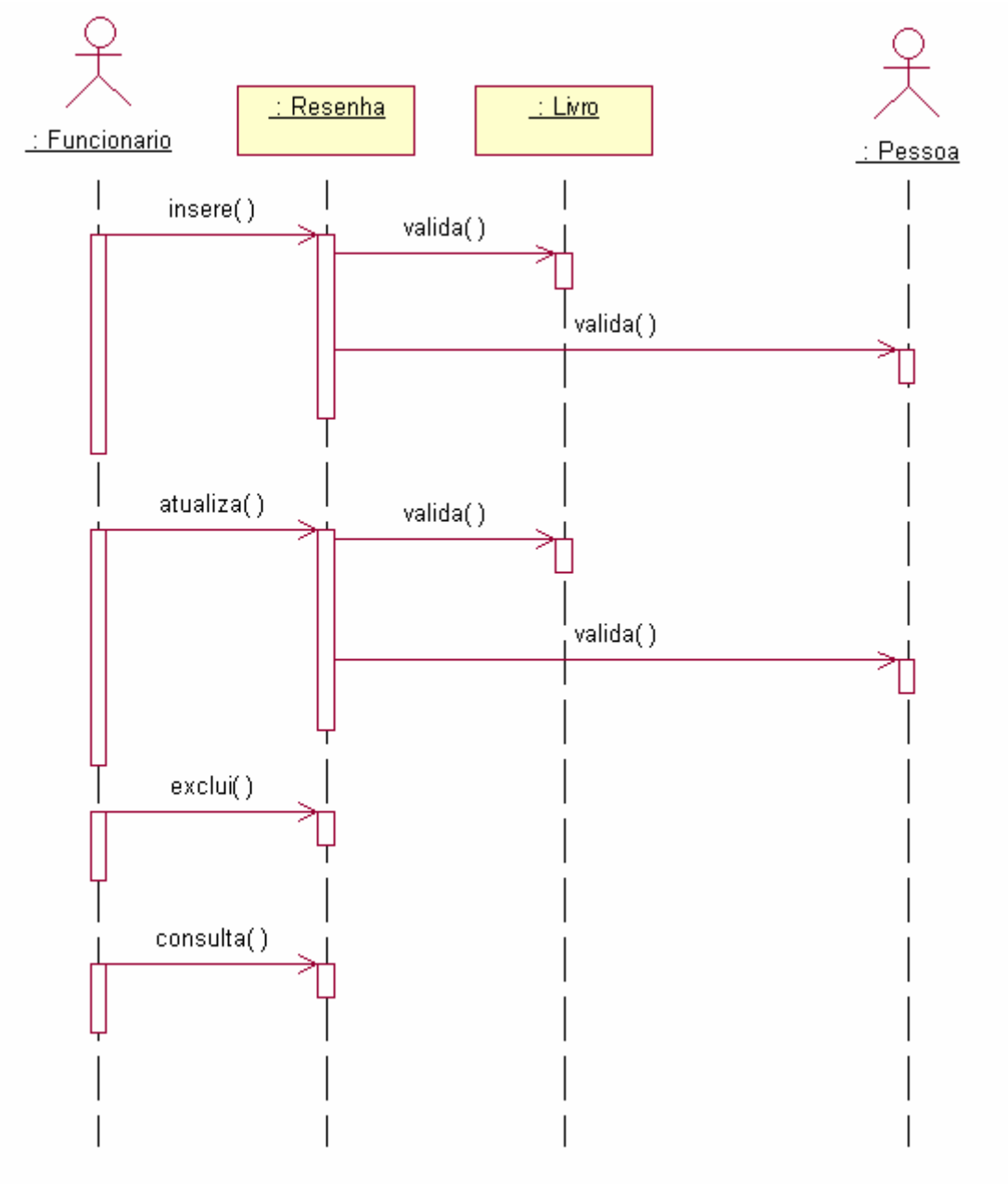
b) **Manter Curso**: Permite consultar, inserir, alterar e excluir cursos. Um curso pode ser excluído somente se nenhum aluno está matriculado naquele curso. A Figura 14 representa o diagrama de seqüência deste caso de uso;

Figura 14: Diagrama de seqüência - Manter Curso



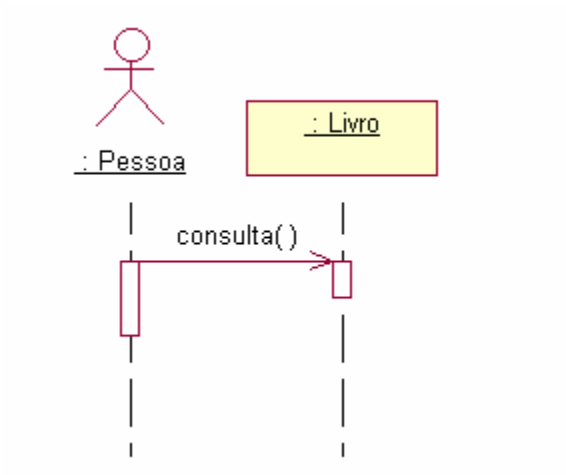
c) **Manter Resenha**: Permite consultar, inserir, alterar e excluir resenhas. Estas funcionalidades somente são disponíveis aos funcionários que utilizam o sistema interno. A Figura 15 representa o diagrama de seqüência deste caso de uso;

Figura 15: Diagrama de seqüência - Manter Resenha



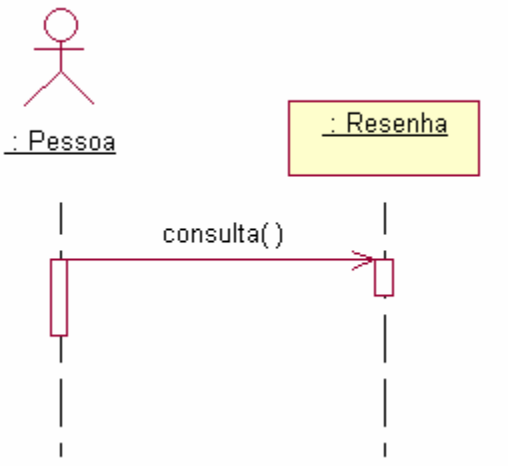
d) **Consultar Livros**: A pessoa pode informar critérios de pesquisa ou não e executar uma consulta. Deve ser retornada uma lista de livros que atendem aos critérios definidos. Os critérios de pesquisa possíveis são: Código do Livro, Código ISBN, Classificação, Nome do Autor e Título do Livro. A Figura 16 representa o diagrama de seqüência deste caso de uso;

Figura 16: Diagrama de seqüência - Consultar Livros



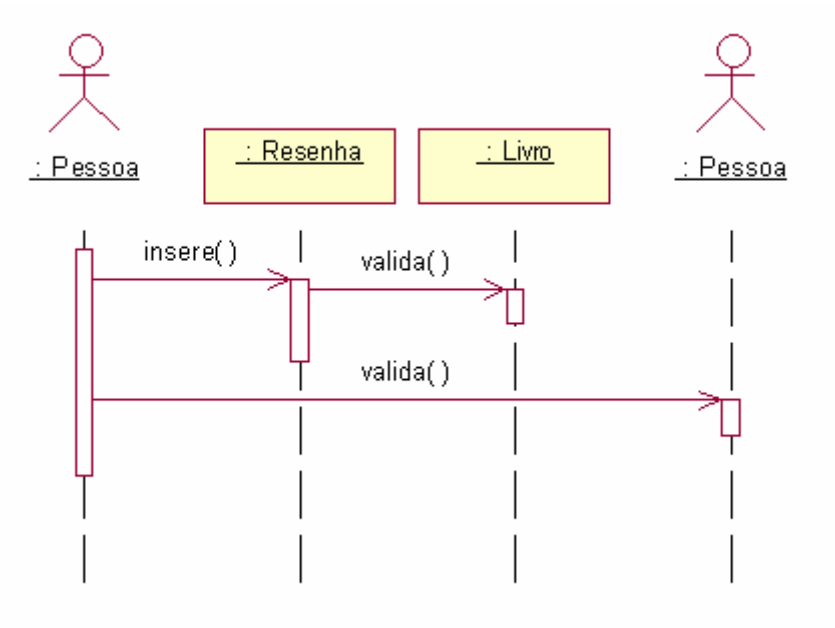
e) **Consultar Resenhas**: A pessoa pode informar critérios de pesquisa ou não e executar uma consulta. Deve ser retornada uma lista de resenhas que atendem aos critérios definidos. Os critérios de pesquisa possíveis são: Código do Livro e Código do Autor. A Figura 17 representa o diagrama de seqüência deste caso de uso;

Figura 17: Diagrama de seqüência - Consultar Resenhas



f) **Escrever Resenha**: Permite somente inserir resenhas. Esta funcionalidade deve estar disponível somente pela Internet. A Figura 18 representa o diagrama de seqüência deste caso de uso.

Figura 18: Diagrama de seqüência - Escrever Resenha



Os casos de uso secundários são:

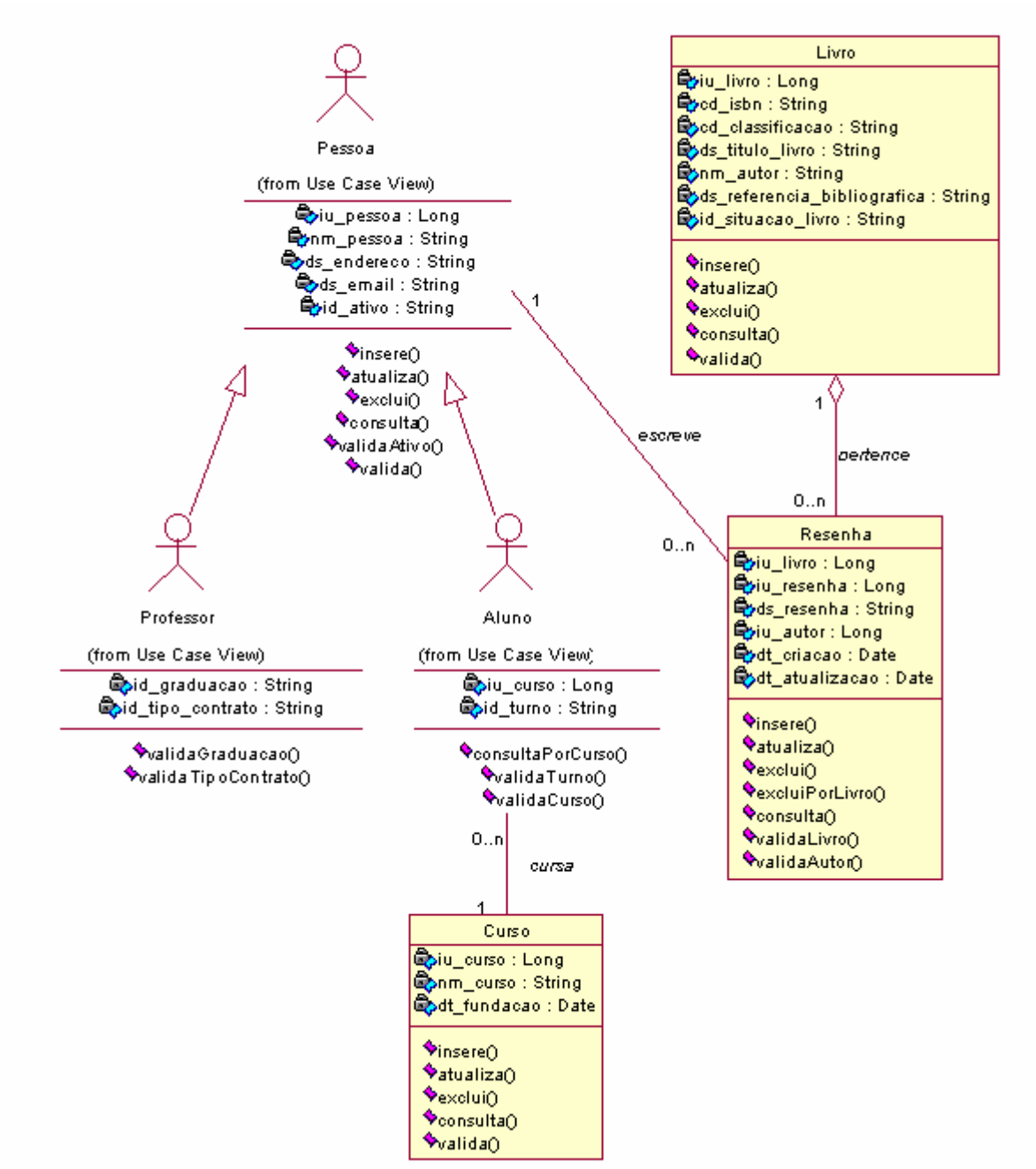
a) **Manter Professor** que permite consultar, inserir e alterar professores. Não é possível excluir um professor, somente desativá-lo. Estas funcionalidades somente são disponíveis aos funcionários que utilizam o sistema interno (aplicação Java);

b) **Manter Aluno** que permite consultar, inserir e alterar alunos. Não é possível excluir um aluno, somente desativá-lo. Estas funcionalidades somente são disponíveis aos funcionários que utilizam o sistema interno (aplicação Java).

5.2.1.2 DIAGRAMA DE CLASSES

A partir do diagrama de casos de uso, o diagrama de classes é modelado conforme a Figura 19.

Figura 19: Diagrama de classe



5.2.2 IMPLEMENTAÇÃO

Com o diagrama de classes é possível gerar no Rational Rose um arquivo IDL contendo as definições de todos os objetos da aplicação, necessário para a implementação da

aplicação em CORBA. Porém, para o protótipo todas as definições IDL foram produzidas manualmente porque ocorreram diversos erros no momento da geração das definições IDL. Por exemplo, os tipos de dados dos atributos e métodos não eram reconhecidos pelo Rational Rose como tipos de dados definidos pelo padrão CORBA. O Quadro 4 exemplifica o código IDL criado para a classe Curso conforme o diagrama de classes exibido na Figura 19.

Quadro 4: Definição IDL para a classe Curso

```
//
module tcc {

  interface Curso {
    //
    // atributos
    readonly attribute long iu_curso;
    attribute string nm_curso;
    //
    // metodos
    boolean insere(in long iu_curso, in string nm_curso);
    boolean atualiza(in long iu_curso, in string nm_curso);
    boolean exclui(in long iu_curso);
    long consulta(in long iu_curso, in string nm_curso);
    boolean valida(in long iu_curso);

  };
};
```

Conforme explanado na seção 3.2 é necessário compilar o arquivo IDL. A compilação do arquivo “Curso.idl”, que contém as construções listadas no Quadro 4, se dá através da linha de comando:

```
C:\JAVA org.openorb.compiler.IdlCompiler Curso <ENTER>
```

Os arquivos gerados pela linha de comando acima são:

a) Curso.java;
b) CursoHelper.java;
c) CursoHolder.java;
d) CursoOperations.java;
e) _CursoStub.java;
f) CursoPOA.java;
g) CursoPOATie.java.

Após a compilação destes arquivos em Java, estes objetos estão prontos para serem utilizados pela a implementação do objeto Curso como apoio para a sua comunicação com o ORB.

5.2.3 OPERACIONALIDADES DA IMPLEMENTAÇÃO

A aplicação é constituída de três módulos desenvolvidos de duas formas diferentes: o servidor (ORB), a maioria dos clientes são aplicativos Java e um cliente é *applet* Java.

O primeiro serviço que deve estar disponível (executando) é o serviço de nomes. Com o serviço de nomes disponível, o ORB pode ser iniciado e registrar seus objetos no serviço de nomes para que os clientes possam acessar esses objetos. O serviço de nomes utilizado já vem embutido no OpenORB (Daniel, 2001). A linha de comando abaixo inicia o serviço de nomes na porta 2001 do servidor:

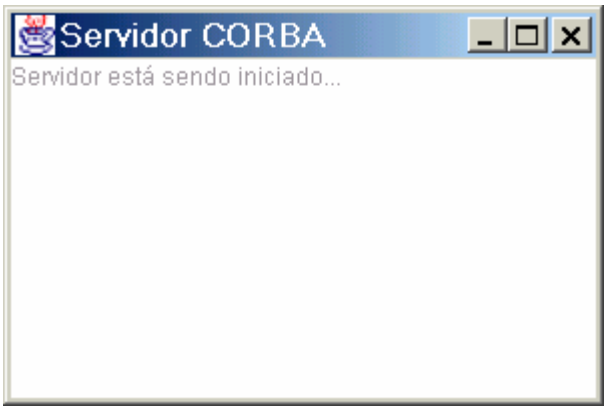
```
C:\JAVA org.openorb.util.MapNamingContext –ORBPort=2001 <ENTER>
```

A seguir, o ORB deve ser iniciado. O ORB inicia o ambiente, cria os objetos de implementação, disponibiliza os objetos e serviços para os clientes e fica a espera que eventos sejam gerados por clientes. Geralmente, estes procedimentos não são executados por um usuário final. O seguinte comando é executado para iniciar o ORB:

```
C:\JAVA tcc.Servidor <ENTER>
```

A Figura 20 exibe o servidor sendo executado.

Figura 20: Tela do servidor CORBA



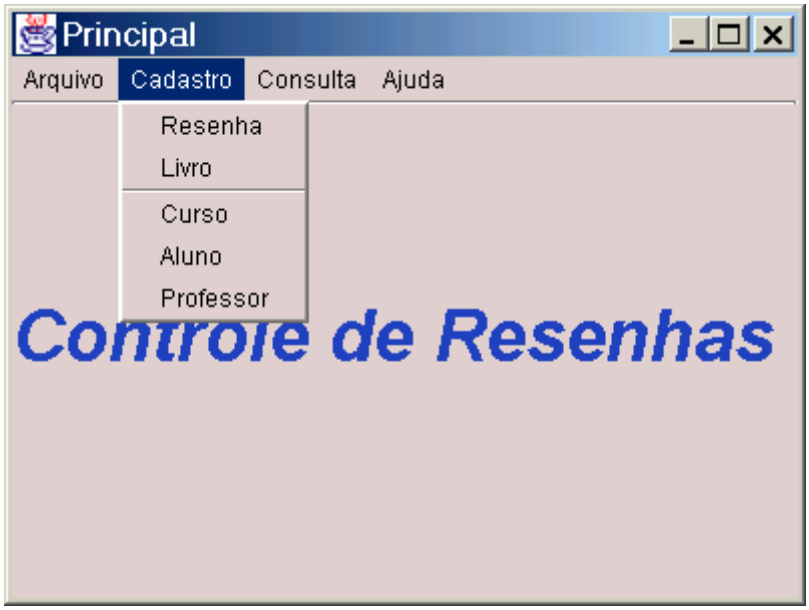
Enquanto o ORB é iniciado, são iniciados os serventes que constituem o serviço de persistência. Após o ORB ser iniciado, os clientes já podem acessá-lo, seja pelo aplicativo Java ou pelo *applet* Java.

Para executar a aplicação Java, a seguinte linha de comando deve ser digitada:

```
C:\JAVA tcc.Aplicação <ENTER>
```

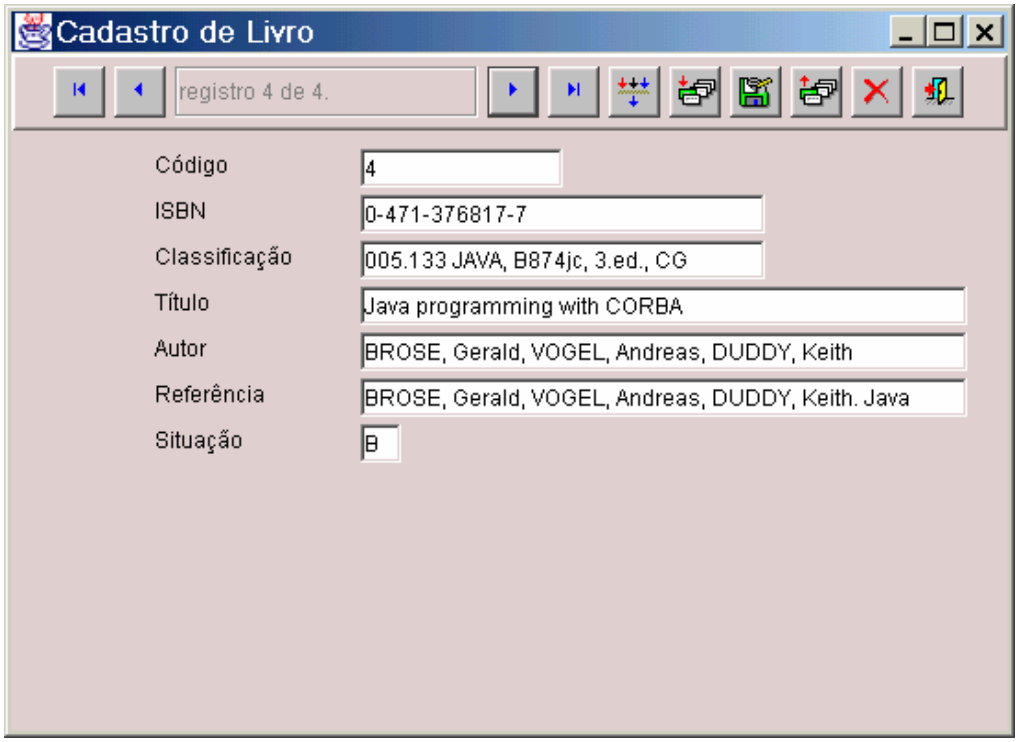
A Figura 21 demonstra a tela de entrada do aplicativo.

Figura 21: Tela de entrada da aplicação



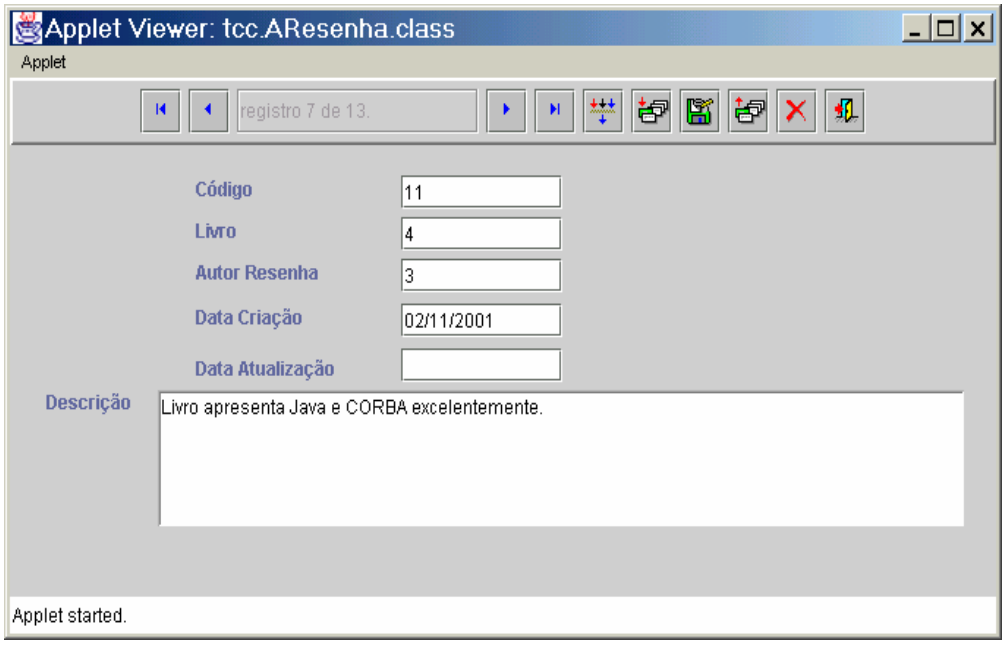
A aplicação pode ser executada de qualquer ponto da rede. A Figura 22 exemplifica uma tela do protótipo sendo utilizada.

Figura 22: Tela de cadastro de livro.



Também está disponível um *applet* Java para a inclusão de resenhas. A Figura 23 ilustra o *applet* sendo utilizado.

Figura 23: Applet para escrever resenhas.



5.3 SERVIÇO DE PERSISTÊNCIA

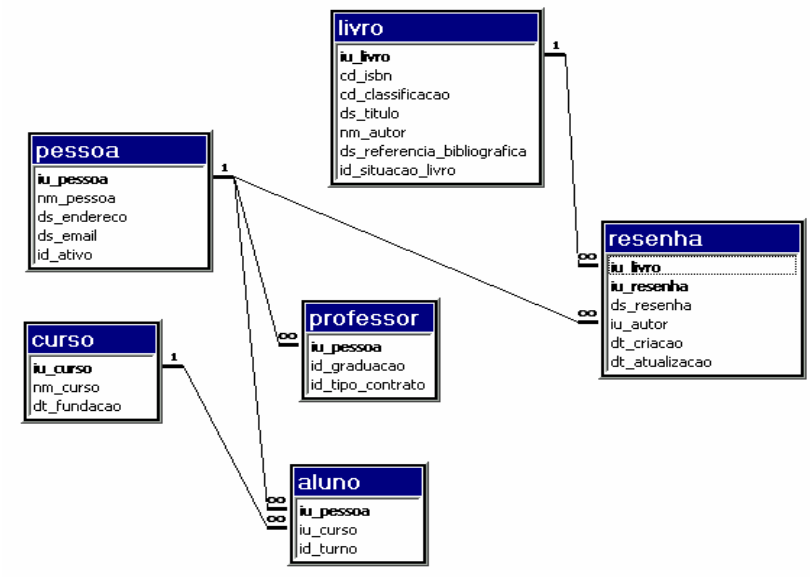
Com a abordagem do protótipo de aplicação, a descrição do desenvolvimento do serviço de persistência pode ser compreendida de uma maneira mais fácil, devido ao fato do serviço de persistência empregado ser transparente. O serviço de persistência tem como principal objetivo armazenar os dados dos objetos para execuções posteriores da aplicação.

A persistência será implementada de tal forma que os objetos desta aplicação estejam concentrados em um nodo da rede, local este em que os objetos serão mais utilizados, por exemplo na biblioteca. Porém, nada impede que os objetos estejam distribuídos na rede. Outros sistemas podem acessar estas informações de outros nodos da rede, caracterizando os sistemas de Objetos Distribuídos.

5.3.1 ESPECIFICAÇÃO

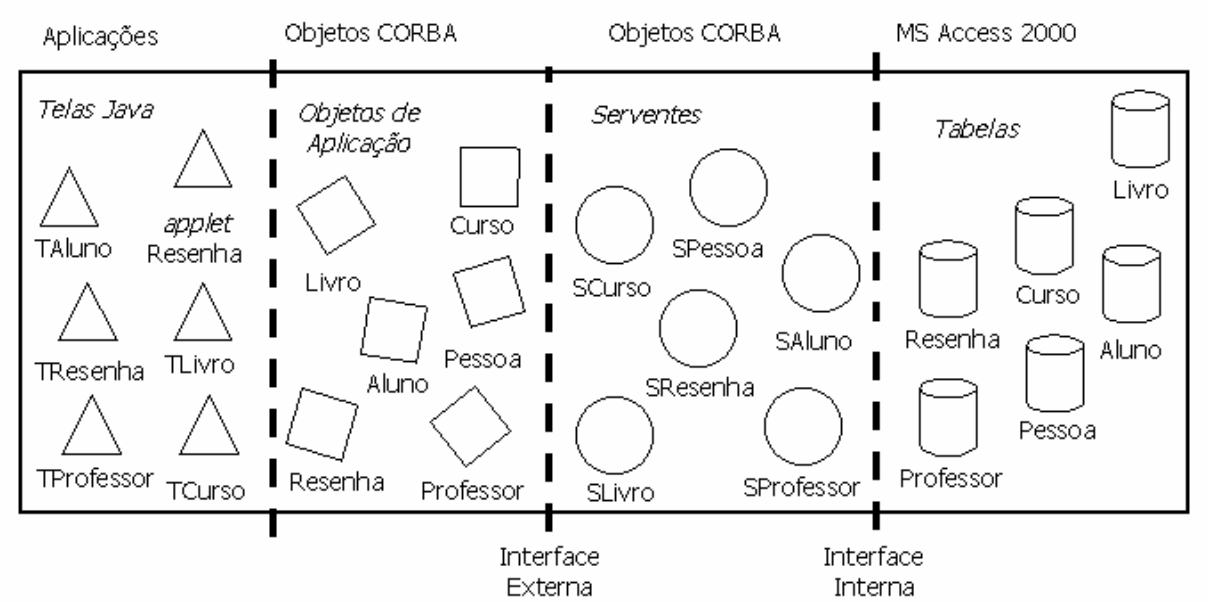
Para cada classe concreta do diagrama de classes do protótipo da aplicação deve existir uma tabela correspondente no banco de dados. Cada registro da tabela representa um objeto daquela classe. Para armazenar os objetos persistentes no banco de dados, é realizada a derivação que consta na seção 4.1.2.2 manualmente. A derivação do diagrama de classes da aplicação resultou em um esquema de tabelas que consiste na estrutura do *datastore* empregado, conforme a Figura 24.

Figura 24: Estrutura do *datastore* utilizado



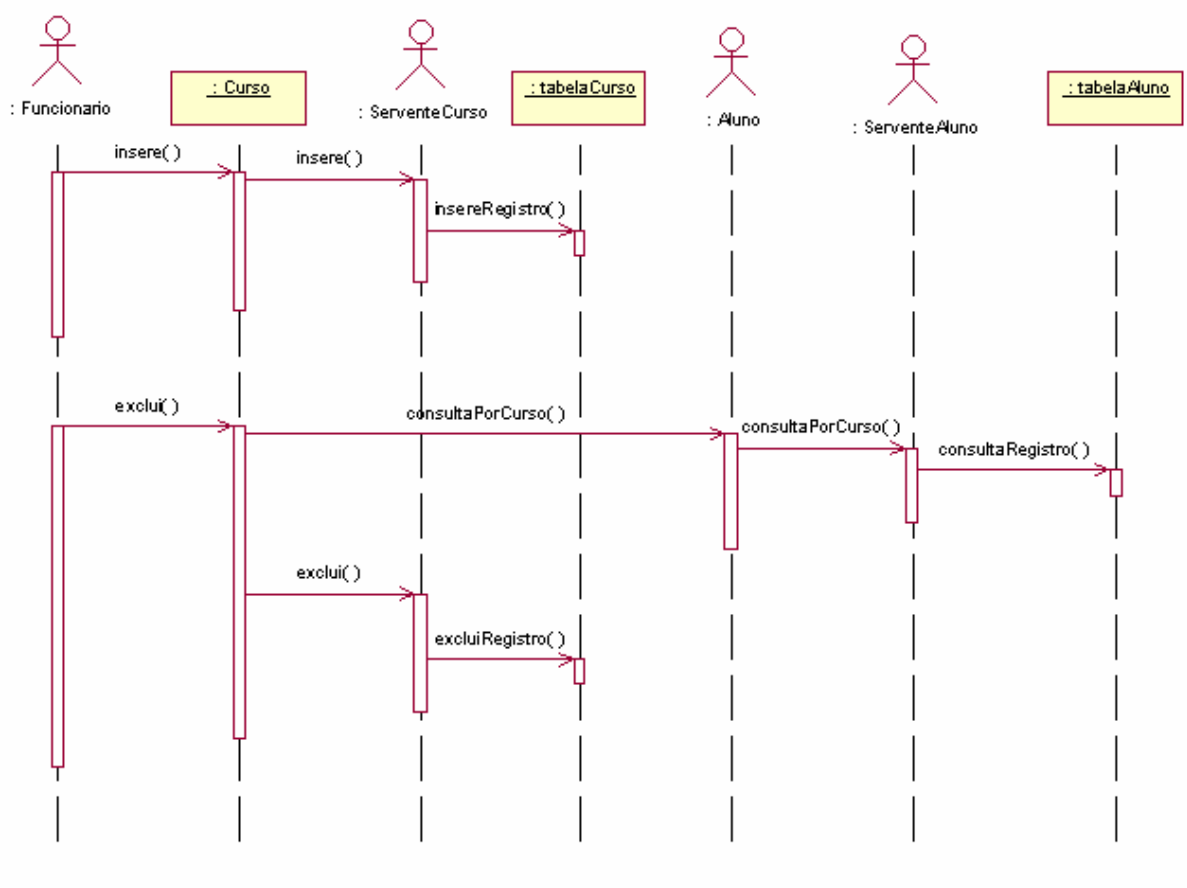
O protótipo de aplicação deve ser incrementado, de modo que permita a comunicação entre os objetos de aplicação e os objetos de armazenamento realizando a persistência dos objetos. Para isso são criados os serventes (*servants*). Para cada tipo de objeto de armazenamento é necessário criar um servente. Os serventes podem trocar mensagens diretamente entre si. Os métodos que os serventes devem possuir são os métodos da classe-base do servente que recuperam ou gravam informações nos objetos. A Figura 25 demonstra a disposição dos objetos na aplicação.

Figura 25: Disposição dos objetos na aplicação



A Figura 26 ilustra como os serventes são empregados, utilizando como exemplo algumas operações do diagrama de seqüência da Figura 14.

Figura 26: Diagrama de seqüência de exemplo do Serviço de Persistência



Descrevendo a operação “insere” da Figura 26, a interface do funcionário possui uma referência ao objeto CORBA “Curso” e envia uma mensagem requisitando a inserção de um curso. Em seguida, o objeto Curso realiza as consistências necessárias, se houverem, e envia uma mensagem ao seu servente requisitando a inserção de um registro na tabela de Curso. O servente faz a conexão com o banco de dados, insere o registro na tabela e desfaz a conexão, confirmando a inclusão no banco de dados.

5.3.2 IMPLEMENTAÇÃO

Para a implementação do serviço de persistência é necessário preparar o ambiente em que os objetos serão persistidos. O ponto inicial foi a criação das tabelas e relacionamentos no banco de dados selecionado de acordo com os resultados da derivação do modelo de objetos para o modelo de tabelas. O método JDBC foi selecionado para acessar o banco de dados em

Java. Sendo assim, é necessária a criação de uma Fonte de Dados ODBC com o nome “biblioteca” que é mapeada para o banco de dados, no caso, o arquivo “biblioteca.mdb” .

A etapa seguinte é gerar os arquivos IDL que contêm as definições IDL dos serventes. O Quadro 5 lista o arquivo IDL que representa o servente da classe Curso.

Quadro 5: Definição IDL para o servente SCurso

```
//
module tcc {

  interface SCurso {
    //
    // atributos

    //
    // metodos
    boolean insere(in long iu_curso, in string nm_curso);
    boolean atualiza(in long iu_curso, in string nm_curso);
    boolean exclui(in long iu_curso);
    boolean consulta(inout long iu_curso, inout string nm_curso);
    long qtdeTotal(in long iu_curso, in string nm_curso);
  };
};
```

Os arquivos gerados pela compilação do arquivo IDL servem para a comunicação dos objetos com o ORB. A implementação do objeto é realizada separadamente. O Anexo 2 lista o código-fonte da implementação em Java para o servente SCurso.

5.4 SÍNTESE DOS RESULTADOS

A seguir, é descrita a síntese do roteiro utilizado para o desenvolvimento de uma aplicação distribuída que utiliza a Persistência Transparente, especificada pelo OMG.

a) realizar a coleta dos dados e especificar a aplicação utilizando os conceitos de orientação a objetos. Neste momento não deve existir preocupação com a distribuição e a persistência dos objetos;
b) definir o nível de persistência de cada objeto, a forma e a estratégia a ser adotada para a persistência dos objetos;
c) desenvolver a forma e a estratégia de persistência adotada;

d) especificar os métodos para armazenar e recuperar os objetos de cada classe definida;
e) especificar e implementar os serventes de cada classe definida;
f) implementar as classes definidas, inclusive os métodos para armazenar e recuperar os objetos, enviando requisições aos seus respectivos serventes;
g) implementar a aplicação;
h) realizar um estudo para a distribuição dos objetos e distribuir os objetos na rede;

6 CONCLUSÕES

O paradigma de Orientação a Objeto está gradativamente ganhando força no mercado a nível mundial. Talvez, o maior impasse para a migração em massa do mercado estruturado para o mercado orientado a objeto seja a falta de uma tecnologia ou a falta de produtos que realizem a persistência dos objetos de uma forma eficiente e eficaz, proporcionando principalmente segurança às informações.

O OMG realiza um árduo trabalho, que é especificar e padronizar todos os conceitos relativos a objetos distribuídos. Cabe ao desenvolvedor seguir ou não estas especificações para a implementação de um sistema de objetos distribuídos.

Os objetivos do trabalho foram atingidos, uma vez que o serviço de persistência implementado demonstra satisfatoriamente o armazenamento dos objetos. Contudo, o serviço de persistência não atende na totalidade as conformidades descritas em sua especificação devido ao curto espaço de tempo para o estudo e confecção deste trabalho.

O Serviço de Persistência através da persistência transparente possui complexidade menor do que através da utilização de PSDL, tanto para o desenvolvimento como para a utilização do serviço. Este fator colabora para que os sistemas de objetos distribuídos que necessitam do serviço de persistência utilizem a persistência transparente.

A flexibilidade que o padrão CORBA oferece é de grande importância. Serviços podem ser incluídos, retirados ou excluídos do ORB sem maiores dificuldades. O OpenORB, por exemplo, permite a configuração de inúmeros itens e possui um aplicativo somente para gerenciar a configuração do ORB.

A linguagem de programação Java mostrou-se eficiente para o desenvolvimento do trabalho. As técnicas e ferramentas utilizadas se comportaram de forma satisfatória. O Rational Rose permite que, através do diagrama de classes, sejam geradas as definições IDL para os objetos CORBA. O JBuilder é uma ferramenta muito funcional e permitiu a construção de telas utilizando o ambiente gráfico, além de gerenciar todos os programas.

Pelo fato da seleção do produto ORB levar em consideração a necessidade da implementação do ORB ser através de Java, não foi possível executar o cliente em uma outra

linguagem a não ser em Java. Para executar um cliente implementado em outra linguagem é necessário um outro ORB implementado na mesma linguagem do cliente. Este ORB pode então estabelecer comunicação com o ORB em Java.

Por fim, como resultado deste trabalho, pode-se afirmar que os conceitos relativos a Objetos Distribuídos deixarão de ser tendências e passarão a ser realidade à medida que os desenvolvedores tomem conhecimento desta tecnologia. O padrão CORBA é atualmente a principal especificação desta tecnologia que amadurece a cada momento.

6.1 EXTENSÕES

Para desenvolvimento futuro, sugere-se a implementação de um ORB que suporte as funções básicas da especificação CORBA, bem como os Serviços Comuns também são ótimos temas para trabalhos futuros. Em especial, o serviço de segurança porque pode ser vital para uma aplicação comercial de sucesso.

Outra sugestão é implementar um compilador PSDL e o módulo *CosPersistentState* para o serviço de persistência.

A forma de representação dos vários tipos de objetos CORBA ou uma ferramenta visual para gerar arquivos IDL ou PSDL é, sem dúvida, uma ótima sugestão para trabalhos futuros, pois existem atualmente dificuldades para a representação de aplicações CORBA.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. **Análise e projeto orientados a objeto**: seu guia para desenvolver sistemas robustos com tecnologia de objetos. Rio de Janeiro : Infobook, 1998.

ARMSTRONG, Eric. **JBuilder 2 bible**. Foster City : IDG Books Wordwide, 1998.

BROSE, Gerald, VOGEL, Andreas, DUDDY, Keith. **Java programming with CORBA**. 3.ed. New York: OMG Press, 2001.

BOOCH, Grady, et al. **UML** : guia do usuário. Rio de Janeiro : Campus, 2000.

CAPELETTO, Johni Jeferson. **Comunicação entre objetos distribuídos utilizando a tecnologia CORBA common object request broker architecture** . 1999. 60 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

COAD, Peter, YOURDON, Edward. **Análise baseada em objetos**. Rio de Janeiro: Campus, 1992.

DANIEL, Jerome, DANIEL, Marina. **The OpenORB project**, [2001?]. Disponível em: <http://www.openorb.org>. Acesso em: 02 nov. 2001.

DEITEL, Harvey M, DEITEL, P. J. **Java** : como programar. 3.ed. Porto Alegre : Bookman, 2001.

INPRISE CORPORATION. **Visibroker for Java 4.5**. Scotts Valley : Inprise, 2000.

IONA. **IONA: e-business platform applications for total business integration**, Dublin, [2001?]. Disponível em: <http://www.iona.com>. Acesso em: 02 ago. 2001.

KHOSHAFIAN, Setrang. **Banco de dados orientado a objetos**. Rio de Janeiro: Infobook, 1994.

MAINETTI, Sergio Jr. Objetos distribuídos: um novo padrão em cliente/servidor. **Developers' magazine**, Rio de Janeiro, v. 2, n. 16, p. 36-42, dezembro. 1997.

MARTIN, James. **Princípios de análise e projeto baseado em objetos.** Rio de Janeiro : Campus, 1994.

MONTEZ, Carlos. **Um modelo de programação para aplicações de tempo real em sistemas abertos**. Monografia do Exame de Qualificação de Doutorado. Florianópolis, DAS: UFSC, 1997.

OMG. **The common object request broker architecture specification**, OMG, set. [2001]. Disponível em: <ftp://ftp.omg.org/pub/docs/formal/01-09-01.pdf>. Acesso em: 19 set. 2001.

OMG. **A discussion of the object management architecture**, OMG, jan. [1997]. Disponível em: <ftp://ftp.omg.org/pub/docs/formal/00-06-41.pdf>. Acesso em: 19 set. 2001.

OMG. **Persistent state service specification**, OMG, ago. [1999]. Disponível em: <ftp://ftp.omg.org/pub/docs/formal/99-07-07.pdf>. Acesso em: 16 set. 2001.

OMG. **The object management group**, OMG, nov. [2001]. Disponível em: <http://www.omg.org>. Acesso em: 12 nov. 2001.

PIRES, Aldo. CORBA: um padrão para objetos distribuídos. **Developers' magazine**, Rio de Janeiro, v. 1, n. 12, p. 36-39, agosto. 1997.

QUEIROZ, José Antônio Monteiro de; CUNHA, Paulo Roberto Freire. **Sistemas distribuídos**: de especificações LOTOS a implementações. Recife : UFPE-DI, 1994.

Rational. **Rational Rose manuals**, ago. [2001]. Disponível em: <http://www.rational.com/support/documentation/manuals/rose.jsp>. Acesso em: 26 ago. 2001.

SIEGEL, Jon, et all. **CORBA fundamentals and programming**. New York: John Wiley & Sons, 1996.

WUTKA, Mark. **Java**: técnicas profissionais**.** São Paulo : Berkeley, 1997.

ANEXO 1 – IDL MÓDULO *COSPERSISTENTSTATE*

```
//File: CosPersistentState.psdl
// Copyright 1998-1999 by the Object Management Group.
// All Rights Reserved.

#ifndef _COS_PERSISTENT_STATE_PSDL_
#define _COS_PERSISTENT_STATE_PSDL_

#include <orb.idl>
#include <CosTransactions.idl>

module CosPersistentState {
   local interface CatalogBase;
   local interface Connector;
   local interface EndOfAssociationCallback;
   local interface Session;
   local interface SessionPool;
   local interface StorageHomeBase;
   local interface TransactionalSession;

   native StorageObjectBase;
   native StorageObjectFactory;
   native StorageHomeFactory;
   native SessionFactory;
   native SessionPoolFactory;

   exception NotFound {};

   typedef string TypeId;
   typedef CORBA::OctetSeq Pid;
   typedef CORBA::OctetSeq ShortPid;

   abstract storagetype StorageObject {
      void destroy_object();
      boolean object_exists();
      Pid get_pid();
      ShortPid get_short_pid();
      StorageHomeBase get_storage_home();
   };

   enum YieldRef { YIELD_REF };
   enum ForUpdate { FOR_UPDATE };
   typedef short IsolationLevel;
   const IsolationLevel READ_UNCOMMITTED = 0;
   const IsolationLevel READ_COMMITTED = 1;
   const IsolationLevel REPEATABLE_READ = 2;
   const IsolationLevel SERIALIZABLE = 3;
   typedef short TransactionPolicy;
   const TransactionPolicy NON_TRANSACTIONAL = 0;
   const TransactionPolicy TRANSACTIONAL = 1;
   typedef short AccessMode;
   const AccessMode READ_ONLY = 0;
   const AccessMode READ_WRITE = 1;

   struct Parameter {
```

```
   string name;
   any val;
};

typedef sequence<Parameter> ParameterList;
typedef sequence<TransactionalSession> TransactionalSessionList;

//--------------------------------------------------------------
// Connector
//--------------------------------------------------------------
local interface Connector {
   readonly attribute string implementation_id;
   Pid get_pid(in StorageObjectBase obj);
   ShortPid get_short_pid(in StorageObjectBase obj);
   Session
   create_basic_session(
      in AccessMode access_mode,
      in TypeId catalog_type_name,
      in ParameterList additional_parameters
   );
   TransactionalSession
   create_transactional_session(
      in AccessMode access_mode,
      in IsolationLevel default_isolation_level,
      in EndOfAssociationCallback callback,
      in TypeId catalog_type_name,
      in ParameterList additional_parameters
   );
   SessionPool
      create_session_pool(
      in AccessMode access_mode,
      in TransactionPolicy tx_policy,
      in TypeId catalog_type_name,
      in ParameterList additional_parameters
   );
   TransactionalSession current_session();
   TransactionalSessionList
   sessions(
      in CosTransactions::Coordinator transaction
   );
   StorageObjectFactory
   register_storage_object_factory(
      in TypeId storage_type_name,
      in StorageObjectFactory factory
   );
   StorageHomeFactory
   register_storage_home_factory(
      in TypeId storage_home_type_name,
      in StorageHomeFactory factory
   );
   SessionFactory
   register_session_factory(
      in TypeId catalog_type_name,
      in SessionFactory factory
   );
   SessionPoolFactory
   register_session_pool_factory(
      in TypeId catalog_type_name,
      in SessionPoolFactory factory
```

```
   );
};

//--------------------------------------------------------------
// CatalogBase
//--------------------------------------------------------------
local interface CatalogBase {
   readonly attribute AccessMode access_mode;
   StorageHomeBase
   find_storage_home(in string storage_home_id)
   raises (NotFound);
   StorageObjectBase
   find_by_pid(in Pid the_pid) raises (NotFound);
   void flush();
   void refresh();
   void free_all();
   void close();
};

//--------------------------------------------------------------
// StorageHomeBase
//--------------------------------------------------------------
local interface StorageHomeBase {
   StorageObjectBase
   find_by_short_pid(in ShortPid short_pid)
   raises (NotFound);
   CatalogBase get_catalog();
};

//--------------------------------------------------------------
// Session
//--------------------------------------------------------------
   local interface Session : CatalogBase {};

//--------------------------------------------------------------
// TransactionalSession
//--------------------------------------------------------------
local interface TransactionalSession : Session {
   readonly attribute IsolationLevel default_isolation_level;
   typedef short AssociationStatus;
   const AssociationStatus NO_ASSOCIATION = 0;
   const AssociationStatus ACTIVE = 1;
   const AssociationStatus SUSPENDED = 2;
   const AssociationStatus ENDING = 3;
   void start(in CosTransactions::Coordinator transaction);
   void suspend(in CosTransactions::Coordinator transaction);
   void end(
      in CosTransactions::Coordinator transaction,
      in boolean success
   );
   AssociationStatus get_association_status();
   CosTransactions::Coordinator transaction();
};

local interface EndOfAssociationCallback {
   void released(in TransactionalSession session);
};

//--------------------------------------------------------------
```

```
   // SessionPool
   //-------------------------------------------------------------
   typedef sequence<Pid> PidList;
   local interface SessionPool : CatalogBase {
      void flush_by_pids(in PidList pids);
      void refresh_by_pids(in PidList pids);
      readonly attribute TransactionPolicy transaction_policy;
   };
};
#endif // _COS_PERSISTENT_STATE_PSDL_
```

ANEXO 2 – CÓDIGO-FONTE DO SERVENTE SCURSO

```
package tcc;

import java.sql.*;

/**
 * Title:        Trabalho Final de Conclusão de Curso
 * Description:  Este projeto tem como objetivo desenvolver um Serviço de
Persistência para o Modelo CORBA.
 * Copyright:    Copyright (c) 2001
 * Company:      FURB
 * @author Rafael Cristiano Macedo
 * @version 1.0
 */

public class SCursoImpl extends SCursoPOA{

  public boolean insere(int iu_curso, String nm_curso) {
    // insere um curso no BD
    String ds_comando;
    Connection conexao;
    Statement stmt;
    int resultado;

    ds_comando = "INSERT INTO curso( ";
    ds_comando += " iu_curso";
    ds_comando += ",nm_curso)";
    ds_comando += " VALUES ( ";
    ds_comando += "" + iu_curso + "";
    ds_comando += ",'" + nm_curso + "')";
    // conecta ao BD, insere o curso e desconecta do BD
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      conexao = DriverManager.getConnection("jdbc:odbc:biblioteca");
      conexao.setAutoCommit(true);
      stmt = conexao.createStatement();
      resultado = stmt.executeUpdate(ds_comando);
      conexao.close();
      if (resultado == 1)
        return true;
      else
        return false;
    }
    catch (ClassNotFoundException cnfex) {
      System.err.println("Erro ao abrir conexão com o Banco de Dados.");
      System.err.println("Erro ao inserir Curso.");
      return false;
    }
    catch (SQLException s) {
      System.err.println("Erro ao inserir Curso.");
      return false;
    }
```

```
}

public boolean atualiza(int iu_curso, String nm_curso) {
  // atualiza o curso pelo iu_curso no BD
  String ds_comando;
  Connection conexao;
  Statement stmt;
  int resultado;

  ds_comando = "UPDATE curso SET ";
  ds_comando += " nm_curso = '" + nm_curso + "' ";
  ds_comando += " WHERE iu_curso = " + iu_curso;
  // conecta ao BD, atualiza o curso e desconecta do BD
  try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    conexao = DriverManager.getConnection("jdbc:odbc:biblioteca");
    conexao.setAutoCommit(true);
    stmt = conexao.createStatement();
    resultado = stmt.executeUpdate(ds_comando);
    conexao.close();
    if (resultado == 1)
      return true;
    else
      return false;
  }
  catch (ClassNotFoundException cnfex) {
    System.err.println("Erro ao abrir conexão com o Banco de Dados.");
    System.err.println("Erro ao atualizar Curso.");
    return false;
  }
  catch (SQLException s) {
    System.err.println("Erro ao atualizar Curso.");
    return false;
  }
}

public boolean exclui(int iu_curso) {
  // exclui o curso pelo iu_curso no BD
  String ds_comando;
  Connection conexao;
  Statement stmt;
  int resultado;

  ds_comando = "DELETE FROM curso ";
  ds_comando += " WHERE iu_curso = " + iu_curso;
  // conecta ao BD, exclui o curso e desconecta do BD
  try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    conexao = DriverManager.getConnection("jdbc:odbc:biblioteca");
    conexao.setAutoCommit(true);
    stmt = conexao.createStatement();
    resultado = stmt.executeUpdate(ds_comando);
    conexao.close();
    if (resultado == 1)
      return true;
    else
      return false;
  }
  catch (ClassNotFoundException cnfex) {
```

```
      System.err.println("Erro ao abrir conexão com o Banco de Dados.");
      System.err.println("Erro ao excluir Curso.");
      return false;
    }
    catch (SQLException s) {
      System.err.println("Erro ao excluir Curso.");
      return false;
    }
  }

  public boolean consulta(org.omg.CORBA.IntHolder iu_curso,
org.omg.CORBA.StringHolder nm_curso) {
    // executa a consulta no BD, conforme os parâmetros informados
    String ds_comando;
    Connection conexao;
    Statement stmt;
    ResultSet tabela;
    boolean exec;

    // monta a consulta
    ds_comando = "SELECT * FROM curso WHERE ";
    if (iu_curso.value > 0)
      ds_comando += " (iu_curso = " + iu_curso.value + ") AND ";
    if (!nm_curso.equals(""))
      ds_comando += " (nm_curso = '" + nm_curso.value + "') AND ";
    ds_comando += " (1 = 1) ";
    // conecta ao BD, executa a consulta e desconecta do BD
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      conexao = DriverManager.getConnection("jdbc:odbc:biblioteca");
      conexao.setAutoCommit(true);
      stmt = conexao.createStatement();
      tabela = stmt.executeQuery(ds_comando);
      exec = tabela.next();
      if (exec) {
        iu_curso.value = tabela.getInt("iu_curso");
        nm_curso.value = tabela.getString("nm_curso");
        conexao.close();
        return true;
      }
      else {
        conexao.close();
        return false;
      }
    }
    catch (ClassNotFoundException cnfex) {
      System.err.println("Erro ao abrir conexão com o Banco de Dados.");
      System.err.println("Erro ao executar a consulta em Curso.");
      return false;
    }
    catch (SQLException s) {
      System.err.println("Erro ao executar a consulta em Curso.");
      return false;
    }
  }

  public int qtdeTotal(int iu_curso, String nm_curso) {
    // conta quantos cursos existem no BD, conforme os parâmetros
informados
```

```
    String ds_comando;
    Connection conexao;
    Statement stmt;
    ResultSet tabela;
    boolean exec;

    // monta a consulta
    ds_comando = "SELECT Count(iu_curso) AS qt_registro FROM curso WHERE ";
    if (iu_curso > 0)
      ds_comando += " (iu_curso = " + iu_curso + ") AND ";
    if (!nm_curso.equals(""))
      ds_comando += " (nm_curso = '" + nm_curso + "') AND ";
    ds_comando += " (1 = 1) ";
    // conecta ao BD, conta os cursos e desconecta do BD
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      conexao = DriverManager.getConnection("jdbc:odbc:biblioteca");
      conexao.setAutoCommit(true);
      stmt = conexao.createStatement();
      tabela = stmt.executeQuery(ds_comando);
      exec = tabela.next();
      conexao.close();
      if (exec) {
        return tabela.getInt("qt_registro");
      }
      else {
        return 0;
      }
    }
    catch (ClassNotFoundException cnfex) {
      System.err.println("Erro ao abrir conexão com o Banco de Dados.");
      System.err.println("Erro ao contar Cursos.");
      return -1;
    }
    catch (SQLException s) {
      System.err.println("Erro ao contar Cursos.");
      return -1;
    }
  }
}
```