

**UNIVERSIDADE REGIONAL DE BLUMENAU**  
**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**  
**CURSO DE CIÊNCIAS DA COMPUTAÇÃO**  
(Bacharelado)

**PROTÓTIPO DE SOFTWARE PARA GERAÇÃO DE  
SISTEMAS DISTRIBUÍDOS UTILIZANDO *DESIGN  
PATTERNS***

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

**FABIANO OSS**

BLUMENAU, DEZEMBRO/2001

2001/2-22

# **PROTÓTIPO DE SOFTWARE PARA GERAÇÃO DE SISTEMAS DISTRIBUÍDOS UTILIZANDO *DESIGN PATTERNS***

**FABIANO OSS**

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. Everaldo Artur Grahl — Orientador na FURB

---

Prof. José Roque Voltolini da Silva — Coordenador do TCC

**BANCA EXAMINADORA**

---

Prof. Everaldo Artur Grahl

---

Prof. Paulo César Rodacki Gomes

---

Prof. Maurício Capobianco Lopes

## **AGRADECIMENTOS**

Agradeço, principalmente, aos meus pais, Rosalino e Metilde da Silva Oss, por todo o seu amor, companheirismo e dedicação.

A meus irmãos Gima, Beto e Luís pelo incentivo neste tempo que dediquei a minha graduação.

A minha namorada Vanessa Marylin Luchtenberg por estar presente nos momentos mais importantes da minha vida.

Agradecimentos aos meus amigos que incentivaram na caminhada em busca deste título.

A todos meus professores, principalmente ao professor Everaldo Artur Grahl que com sabedoria e paciência soube me guiar pelos caminhos tortuosos trilhados por este trabalho. E ao professor Paulo Rodacki Gomes pela ajuda na execução do trabalho.

## RESUMO

Este trabalho tem o objetivo de efetuar uma pesquisa da área de engenharia de software distribuída sobre *Design Patterns* para sistemas distribuídos. Para isso faz-se o uso da ferramenta CASE Rational Rose 2000 para a modelagem de sistemas. Como resultado desta pesquisa foi desenvolvido um protótipo para geração de sistemas distribuídos utilizando alguns *Design Patterns* identificados para este trabalho. Os padrões utilizados foram: *distributed callback*, *lock*, notificação, *factory*, *persistent layer*, e *template*. Como arquitetura de sistemas distribuídos foi utilizado o padrão CORBA e para implementação do trabalho foi utilizado o ambiente de desenvolvimento Delphi 6.0.

## **ABSTRACT**

This work has the objective to effect a research in the area of software engineering distributed on Design Patterns. For this is used the tool CASE Rational Rose 2000 for the systems of modeling. As result this research was developed an prototype for generation of systems distributed using some Design Patterns identified for this work. The used Design Patterns had been: distributed callback, lock, notification, factory, persistent to layer, and template. As architecture of distributed systems standard CORBA was used and for work implementation tool Delphi 6.0 was used.

# SUMÁRIO

AGRADECIMENTOS .....	III
RESUMO .....	IV
ABSTRACT .....	V
LISTA DE FIGURAS .....	IX
LISTA DE QUADROS .....	XI
1 INTRODUÇÃO .....	1
1.1 OBJETIVOS.....	3
1.2 ORGANIZAÇÃO.....	4
2 SISTEMAS DISTRIBUÍDOS .....	5
2.1 INTRODUÇÃO.....	5
2.2 PADRÃO CORBA.....	6
2.2.1 VISÃO GERAL .....	6
2.2.2 FUNCIONAMENTO.....	8
2.2.3 ARQUITETURA .....	9
2.2.4 IDL E INTERFACES .....	16
2.2.5 A INTEROPERABILIDADE ENTRE ORBS.....	19
3 <i>DESIGN PATTERNS</i> .....	22
3.1 INTRODUÇÃO.....	22
3.1.1 ORIGENS .....	24
3.1.2 ESTRUTURA DE UM <i>DESIGN PATTERN</i> .....	25
3.1.3 COMO UTILIZAR UM <i>DESIGN PATTERN</i> .....	25
3.1.4 COMO SELECIONAR UM <i>DESIGN PATTERN</i> .....	26
3.1.5 COMO DESCREVER UM <i>DESIGN PATTERN</i> .....	27

3.2	DESCRIÇÃO DOS DESIGN PATTERNS .....	28
4	EXTENSIVE MARKUP LANGUAGE (XML).....	42
4.1	PONTOS FORTES DA XML.....	42
4.2	OBJETIVOS.....	43
4.3	DOCUMENTOS .....	44
4.3.1	ELEMENTOS .....	45
4.3.2	ATRIBUTOS .....	45
4.3.3	REFERÊNCIAS A ENTIDADES.....	45
4.3.4	COMENTÁRIOS .....	46
4.3.5	INSTRUÇÕES DE PROCESSAMENTO .....	46
4.3.6	SEÇÕES CDATA .....	46
4.3.7	DEFINIÇÃO DE TIPO DE DOCUMENTO .....	47
5	FERRAMENTAS .....	48
5.1	DELPHI 6.0.....	48
5.2	KYLIX 2.....	48
5.3	VISIBROKER .....	49
5.3.1	EXEMPLO DE UM PROGRAMA DISTRIBUÍDO.....	50
6	DESENVOLVIMENTO .....	55
6.1	REQUISITOS PRINCIPAIS DO PROBLEMA .....	55
6.2	ESPECIFICAÇÃO DO PROTÓTIPO .....	55
6.2.1	DIAGRAMA DE CASOS DE USO .....	55
6.2.2	DIAGRAMA DE CLASSES .....	57
6.2.3	DIAGRAMA DE SEQÜÊNCIA.....	62
6.3	IMPLEMENTAÇÃO .....	72
6.3.1	DICIONÁRIO .....	73

6.3.2 <i>DESIGN PATTERNS</i> .....	76
6.3.3 GERAR SISTEMAS DISTRIBUÍDOS .....	80
7 CONCLUSÕES .....	91
7.1 PROBLEMAS E LIMITAÇÕES .....	92
7.2 EXTENSÕES .....	92
REFERÊNCIAS BIBLIOGRÁFICAS .....	93
ANEXO 1 – IMPLEMENTAÇÃO DE UM OBJETO DE NEGÓCIO GERADO.....	96
ANEXO 2 – PROGRAMA PRINCIPAL DA APLICAÇÃO CLIENTE .....	99
ANEXO 3 – <i>UNIT</i> DO PROGRAMA CLIENTE .....	100



## LISTA DE FIGURAS

Figura 1 Arquitetura OMA .....	7
Figura 2 Estrutura CORBA .....	10
Figura 3 Uma requisição enviada através do ORB.....	11
Figura 4 Estrutura de um ORB .....	12
Figura 5 Independência de linguagem de programação .....	16
Figura 6 Diagrama de classes do padrão <i>lock</i> .....	32
Figura 7 <i>Pattern de</i> notificação .....	34
Figura 8 Diagrama de classes do padrão <i>factory</i> .....	36
Figura 9 Diagrama de classes do <i>design pattern persisten layer</i> .....	38
Figura 10 Aplicação exemplo.....	54
Figura 11 Diagrama de casos de uso .....	57
Figura 12 <i>Packages</i> .....	57
Figura 13 Diagrama de classes da <i>package</i> dicionário.....	58
Figura 14 <i>Package patterns</i> .....	60
Figura 15 <i>Package</i> geração.....	61
Figura 16 Diagrama de Seqüência ler arquivo UML .....	63
Figura 17 Diagrama de seqüência aplicar <i>Design Patterns</i> .....	66
Figura 18 Diagrama de seqüência gerar IDL.....	67
Figura 19 Diagrama de seqüência gerar DDL.....	68
Figura 20 Diagrama de seqüência compilar IDL.....	69
Figura 21 Diagrama de seqüência gerar <i>Business Server</i> .....	70
Figura 22 Diagrama de seqüência gerar cliente.....	71
Figura 23 Salvar XML.....	72

Figura 24 Diagrama de classes do estudo de caso.....	73
Figura 25 Tela principal dicionário .....	74
Figura 26 Aplicar <i>Design Pattern</i> .....	77
Figura 27 IDL gerada para o servidor .....	78
Figura 28 IDL gerada para o cliente.....	79
Figura 29 IDL gerada com o padrão <i>persistent layer</i> .....	79
Figura 30 Arquitetura de sistemas distribuídos .....	81
Figura 31 <i>DataBase Server</i> .....	82
Figura 32 Gerar DDL .....	83
Figura 33 Tela do Gerar Business Server.....	84
Figura 34 Compilando o servidor de negócios.....	87
Figura 35 Servidor de negócio executando .....	87
Figura 36 Gerando uma aplicação cliente .....	88
Figura 37 <i>Interface</i> Html .....	89
Figura 38 <i>Interface</i> com formulários.....	90

## LISTA DE QUADROS

Quadro 1 Declaração de interface .....	17
Quadro 2 Sintaxe de uma operação .....	18
Quadro 3 Interface <i>callback</i> .....	30
Quadro 4 Exemplo do padrão <i>lock</i> .....	32
Quadro 5 Exemplo do <i>Pattern</i> notificação .....	34
Quadro 6 Sintaxe do serviço de eventos.....	35
Quadro 7 IDL do <i>Design Pattern persistent layer</i> .....	38
Quadro 8 Exemplo do <i>Design Pattern Template</i> .....	40
Quadro 9 Exemplo de XML .....	44
Quadro 10 Exemplo de atributo .....	45
Quadro 11 IDL de exemplo .....	50
Quadro 12 Compilador IDL .....	50
Quadro 13 Exemplo de uma implementação de um objeto.....	51
Quadro 14 Programa principal .....	51
Quadro 15 Aplicação cliente .....	52
Quadro 16 Arquivo do Rational Rose 2000 .....	63
Quadro 17 Especificação do leitor de arquivos do Rational Rose .....	65
Quadro 18 Arquivo XML do dicionário.....	75
Quadro 19 DTD do salvar arquivo em XML .....	76
Quadro 20 Programa principal gerado.....	84
Quadro 21 <i>Unit</i> principal gerada .....	85

# 1 INTRODUÇÃO

Com o aumento da complexidade dos sistemas, desenvolvê-los tornou-se uma tarefa muito difícil. Um dos fatores que gera esta dificuldade é que muitas vezes o entendimento do problema não está muito claro. Além disso, há uma escassez grande na documentação dos problemas e nas soluções encontradas para solucioná-los (Cagnin, 1999). Isso acarreta problemas que muitas vezes se repetem e geram esforços adicionais para a implementação de suas soluções. As tentativas de reuso das boas soluções e do acúmulo de experiências sobre determinados problemas são, na maioria das vezes, iniciativas isoladas de alguns desenvolvedores.

Estudos mostram que quando especialistas trabalham em um problema particular é raro que inventem novas soluções completamente diferentes das já existentes para resolvê-los. Diferentes soluções de projeto são por eles conhecidas como novos problemas e, freqüentemente lembram de outros similares e reusam as soluções antigas, pensando em pares “solução/problema” (Cagnin, 1999).

Segundo Kroth (2000), o reuso de software é uma abordagem dentro da Engenharia de Software que enfoca o reaproveitamento sistemático das três fases do desenvolvimento de sistemas: análise, projeto e codificação. Atualmente, o reuso apresenta resultados mais concretos na fase de codificação, devido à existência de bibliotecas de classes e de componentes disponíveis em larga escala. A fase de análise tem uma prática de reuso mais restrita, pois não possui tecnologia suficientemente madura.

Uma das técnicas existentes para facilitar o reuso de software é a orientação a objetos (Booch, 1994). Entretanto, a simples adoção da tecnologia de objetos, sem a existência de um padrão de reuso explícito e um processo de desenvolvimento de software orientado ao reuso, provavelmente não fornecerá o sucesso esperado no reuso em larga escala. Em outras palavras, a simples adoção da tecnologia de objetos não traz vantagens de reuso, fazendo-se necessário um processo específico que forneça o reuso de software (Jacobson, 1997).

Outras técnicas que propõem o reuso de uma aplicação incluem padrões de codificação denominados de *framework* e padrões de análise definidos com *Design Patterns*. Um *framework*, segundo Taligent (1999), é uma estrutura de classes inter-relacionadas, que constitui uma implementação inacabada, para um conjunto de aplicações de um domínio. Além de permitir a reutilização de um conjunto de classes, um *framework* também minimiza o

esforço de desenvolvimento de novas aplicações, pois já contém a definição de arquitetura gerada a partir dele bem como, tem predefinido o fluxo de controle da aplicação.

Os *Design Patterns*, chamados de padrões de projetos constituem um novo mecanismo para expressar experiências na elaboração de projetos orientados a objetos. Este padrão deveria descrever um problema que ocorre várias vezes e elaborar uma solução reutilizável para ele, de modo que não necessite se preocupar novamente, com a solução do problema nas novas implementações (Gamma, 1995). Segundo Jacobsen (1999), o padrão *Design Patterns*, gera um registro das experiências passadas, de forma que estes moldes resolvam problemas de projeto específicos e tornam estes projetos mais flexíveis, elegantes e reutilizáveis. A meta desses padrões, é a criação de uma linguagem comum, que permita uma comunicação efetiva no que se refere à troca de experiências sobre problemas e suas soluções. Desta forma, soluções que se aplicaram a situações particulares, podem ser novamente aplicadas em situações semelhantes por outros desenvolvedores, para que os *Design Patterns* não fiquem armazenados em bibliotecas escritas em uma linguagem proprietária, o que impede que esta solução seja utilizada por sistemas heterogêneos. A solução seria a utilização de uma estrutura de documentos que se torna independente do meio de distribuição.

Segundo Laurent (1999), a *Extensive Markup Language* (XML) provê uma linguagem padrão para descrição de dados, que possui uma série de pontos fortes, dentre os quais: extensibilidade; auto descritivo; fácil manutenção; simplicidade; portabilidade; entre outros.

A linguagem XML pode ser utilizada para trocar informações entre as organizações (Marchal, 2000). Desta forma esses modelos não ficam dependentes de um fornecedor específico, o que é positivo quando se pensa em interligar soluções.

Segundo Albert Einstein, a necessidade é o fator gerador ou motivador dos novos paradigmas. Em se tratando de desenvolvimento de sistemas não é diferente. As arquiteturas de desenvolvimento sofreram uma grande evolução desde o início da informática. De arquiteturas Monolíticas ou Centralizadas, para arquiteturas Cliente/Servidor ancoradas pelos Sistemas Gerenciadores de Banco de Dados (SGDB) e atualmente para arquiteturas de Objetos Distribuídos, onde de acordo com Lima (2001), os componentes de um sistema assim construídos não precisam estar localizados necessariamente na mesma máquina.

Dentre as arquiteturas de objetos distribuídos, os mais utilizados atualmente são (Lima, 2001):

- a) *Common Object Request Broker Architecture (CORBA)*;
- b) *Java Remote Method Invocation and Specification (Java/RMI)*;
- c) *Distributed Component Object Model (DCOM)*.

Neste trabalho será utilizada a arquitetura CORBA devido ao fato de ser a únicas entre as todas as apresentadas que permite uma interoperabilidade sobre sistemas distribuídos heterogêneos.

Para que essas técnicas citadas sejam realmente utilizadas com o sucesso esperado é necessário especificá-las em uma linguagem de modelagem de objetos. Para Furlan (1998), a *Unified Modeling Language (UML)* é a linguagem padrão para especificar, visualizar, documentar e construir artefatos de um sistema que pode ser utilizada com todos os processos ao longo do ciclo de desenvolvimento e através de diferentes tecnologias de implementação. Existem diversas ferramentas CASE que possuem suporte a linguagem UML, como é o caso do *Rational Rose*, que será empregada neste trabalho.

## 1.1 OBJETIVOS

O objetivo principal deste trabalho foi construir um protótipo para geração de sistemas distribuídos utilizando as técnicas de *Design Patterns*. Esta geração será baseada num repositório escrito na linguagem XML extraído de uma especificação *UML* da ferramenta CASE Rational Rose 2000.

Os objetivos específicos do trabalho são:

- a) adaptar alguns *Design Patterns* existentes em literatura para abranger aspectos de sistemas distribuídos;
- b) desenvolver um padrão para persistência de dados;
- c) criar um dicionário de classes a partir de um arquivo da ferramenta CASE Rational Rose 2000, que contenha uma especificação UML;
- d) unir as tecnologias XML e CORBA no sistema distribuído gerado;
- e) gerar automaticamente os objetos distribuídos utilizando a biblioteca *Cross Platform (CLX)*, onde as aplicações podem ser compiladas no sistema operacional Windows (Delphi 6) e Linux (Kylix2).

## 1.2 ORGANIZAÇÃO

O trabalho foi estruturado da seguinte maneira:

O segundo capítulo apresenta os Sistemas Distribuídos, o padrão CORBA, suas características, funcionamento e arquitetura. Também apresenta algumas características do *middleware* Visibroker.

No terceiro capítulo são apresentados os conceitos e fundamentos sobre *Design Patterns* e também são demonstrados os padrões utilizados neste trabalho.

O quarto capítulo apresenta os conceitos e fundamentos sobre XML.

O quinto capítulo são apresentadas as ferramentas utilizadas para o desenvolvimento do trabalho.

O sexto capítulo apresenta o desenvolvimento do protótipo. Será demonstrada a análise do sistema com os diagramas de caso de uso, diagramas de seqüências e diagrama de classes bem como todos os passos utilizados para o desenvolvimento do mesmo.

O sétimo capítulo apresenta as considerações finais, conclusões, dificuldades encontradas no decorrer do desenvolvimento do trabalho e sugestões para futuras implementações e extensões deste trabalho.

## 2 SISTEMAS DISTRIBUÍDOS

### 2.1 INTRODUÇÃO

A cada dez ou quinze anos ocorre uma revolução na indústria da computação, que eleva a tecnologia da informação a níveis mais simples de implementação. Foi assim no início da informática, nos anos 50, até o final da década de 70. Naquela época foi presenciado o aparecimento da chamada “primeira onda” da informática, a dos chamados sistemas monolíticos ou centralizados protagonizada pelos computadores *mainframes*. Todos os sistemas construídos eram fortemente interdependentes entre si, o que tornou muito difícil a modelagem e reutilização de dados e informações corporativas. Não havia a possibilidade de compartilhar dados com outros sistemas e portanto, cada aplicação deveria possuir um cópia particular dos seus dados. Estas aplicações monolíticas eram ineficientes, não flexíveis, possuíam custo elevado, sendo que a única forma de escalar um determinado sistema, seria através da compra, no mesmo fornecedor, de um *hardware* com maior capacidade de processamento.

Com a disponibilidade comercial dos Sistemas de Gerenciamento de Banco de Dados (SGBD) e a convergência de tecnologias como redes de computadores, no início dos anos 80, as organizações passaram a modelar as suas informações corporativas e a criar um repositório de dados corporativos, possíveis de serem acessados por múltiplos programas, simplificando desta forma o desenvolvimento e manutenção de sistemas complexos. Era o nascimento da “segunda onda” da informática, a da arquitetura “cliente/servidor”. As aplicações podiam ser particionadas em componentes clientes, os quais implementavam a lógica de apresentação e continham muito da lógica do negócio, e os componentes do lado do servidor contendo a lógica do negócio, na forma de *storage procedures* e *triggers* e também a lógica de acesso aos dados corporativos. Como problemas desta arquitetura pode-se citar: a necessidade de criar a mesma funcionalidade diversas vezes, a difícil reutilização do código, dificuldade em escalar a arquitetura “cliente/servidor” principalmente em aplicações voltadas para internet.

Atualmente, se está no limiar de uma nova era, que se pode chamar de sistemas distribuídos (Phampshire, 2001). Os sistemas distribuídos, são um novo paradigma da computação, o qual permite que os componentes do sistema estejam espalhados em uma rede heterogênea. Estes componentes recebem o nome de objetos distribuídos, que podem



interoperar com outros objetos. Bósio (2000) afirma que um objeto distribuído é basicamente a utilização da tecnologia da orientação a objetos em um ambiente distribuído.

Para que os objetos possam ser comunicar, respeitando a característica da heterogeneidade, é necessário a especificação e desenvolvimento de *middlewares* abertos. Um *middleware*, de acordo com Capeletto (1999), é uma camada de software, residente acima do sistema operacional e do substrato de comunicação, que oferece abstrações de alto nível, com objetivo de facilitar a programação distribuída. As abstrações oferecidas fornecem uma visão uniforme na utilização de recursos heterogêneos existentes nas camadas de sistemas operacionais e redes.

Os sistemas distribuídos apresentam vantagens advindas da distribuição tais como disponibilidade, desempenho, otimização de custos, dentre outras. Entretanto, apresentam também as seguintes características: afastamento, concorrência, falta de estado global, ocorrência de falhas parciais, assincronismo, heterogeneidade, autonomia, evolução e mobilidade (Siegel, 1996).

Para fornecer uma plataforma para construção de sistemas distribuídos, a indústria da informática investiu grandes esforços no sentido de desenvolver padrões que permitissem a interoperabilidade entre os componentes do sistema, levando o surgimento de diversas organizações e consórcios tais como *Object Management Group* (OMG), além das já existentes *International Organization for Standardization* (ISS) e *International Telecommunication Union* (ITU). Preocuparam-se também em estabelecer especificações e recomendações que se cristalizaram como *Reference Model* (Bosio, 2000).

## **2.2 PADRÃO CORBA**

### **2.2.1 VISÃO GERAL**

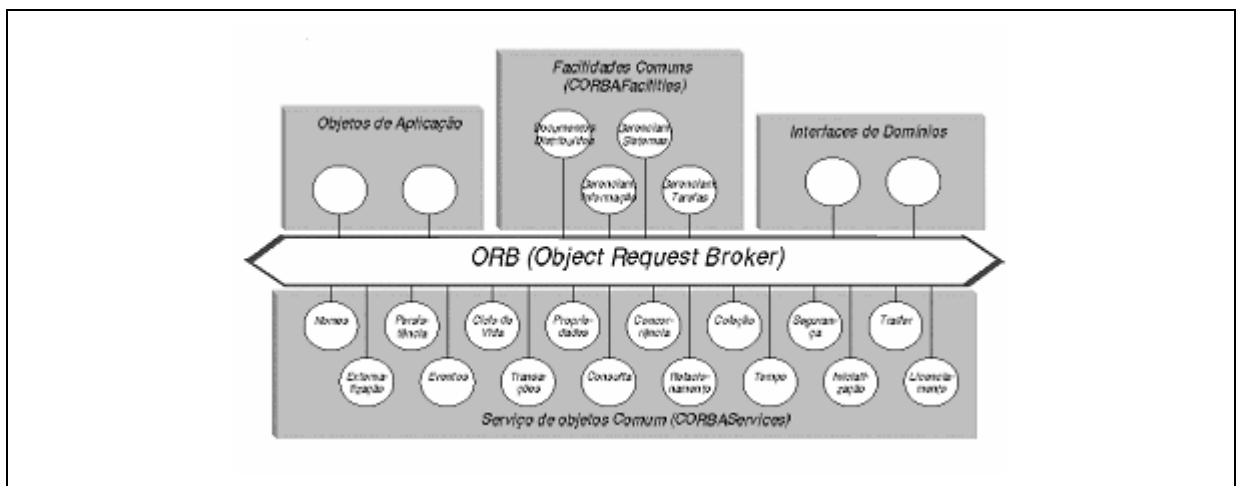
Segundo Brose (2001), o padrão CORBA foi inicialmente implementado em 1991 como sendo um produto intelectual do OMG, um consórcio de mais de 800 companhias das mais diferentes áreas (IBM, Canon, DEC, Philips, Sun, Apple, Informix, Inprise e etc, assim como grandes usuários como Citicorp, British Telecom, American Airlines, e outros), interessadas em prover uma estrutura comum para o desenvolvimento independente de aplicações, usando técnicas de orientação a objeto em redes de computadores heterogêneas.

Ao invés de aplicações, a OMG produz especificações que tornam a computação orientada a objeto possível. Este modelo baseado em objetos permite que métodos de objetos sejam ativados remotamente, através de um elemento intermediário chamado *Object Request Broker* (ORB) situado entre o objeto propriamente dito (que se encontra na camada de aplicação do modelo *Open System Interconnection* (OSI)) e o sistema operacional, acrescido de funcionalidades que o permitam comunicar-se através de rede.

Depois do sucesso inicial, em dezembro de 1994, a OMG lançou o *Internet Inter-ORB Protocol* (IIOP), que se configurou como outro sucesso da OMG (Brose 2001). Antes do IIOP, a especificação CORBA definia somente interação entre objetos distribuídos criados pelo mesmo fornecedor. Os objetos tinham que ser desenvolvidos por uma implementação específica. Usando-se IIOP, a segunda especificação CORBA tornam-se a solução definitiva para a interoperabilidade entre objetos que não estão presos a uma plataforma ou padrão específico.

O CORBA é baseado em pedidos de propostas denominados *Request for Proposals* (RFPs), que relacionam todos os aspectos da tecnologia de objetos, solicitando especificações dos componentes que farão parte de uma arquitetura geral de gestão de objetos, chamado de *Object Management Architecture* (OMA), como pode ser visto na Figura 1.

FIGURA 1 ARQUITETURA OMA



Fonte: Silva (2000)

Os membros da OMG podem então propor uma especificação acompanhada de uma implantação que demonstra os conceitos propostos. Em seguida, a proposta passa por um

processo de apreciação e votação. Este modelo estimulou a indústria da informática e como resultado apareceram um grande número de produtos comerciais (Lima, 2001):

- a) ORBIX (Iona Technologies);
- b) VisiBroker (Inprise);
- c) COOL-ORB (Chorus Systems);
- d) Orblite (HP);
- e) DSOM (IBM);
- f) DOE (Sun);
- g) ISIS;
- h) Orblite;
- i) Orbacus;
- j) OpenDoc (Apple).

O objetivo geral da arquitetura CORBA é permitir a interoperabilidade de objetos de software rodando sobre sistemas distribuídos à priori heterogêneas e permitir sua composição em aplicações.

De acordo com Lima (2001), para garantir a interoperabilidade, a estratégia adotada pela OMG foi:

- a) padronizar a “aparência” externa dos objetos (interface);
- b) propor um mecanismo genérico de ligações entre objetos;
- c) permitir uma programação que esconda a distribuição exata dos objetos (transparências);
- d) oferecer um conjunto de serviços padronizados aos programadores.

## 2.2.2 FUNCIONAMENTO

Para que um objeto cliente requisiere algum serviço de um objeto servidor é necessário que exista uma interface que padronize a aparência externa dos objetos e suas operações. Para resolver este problema a OMG propôs uma linguagem para definição de interfaces, que foi chamada de *Interface Definition Language* (IDL). Ela define as interfaces contratuais de um objeto servidor com seus potenciais clientes.

A única maneira que um objeto cliente pode requisitar um serviço de outro objeto é através de sua interface, ficando o resto dos detalhes do objeto encapsulados. Uma operação sobre um objeto é feita através de uma referência para objeto (ou *Object Reference*). Esta interação é feita pelo *broker* (ORB), fazendo com que os participantes desta interação não precisam se preocupar com detalhes de comunicação, problemas de rede, procura e ativação de objetos, etc. Para o objeto cliente todos os serviços são atendidos de uma forma transparente. Para o objeto distribuído chamado, todas as requisições se comportam da mesma forma, como se fossem requisições locais.

Um objeto distribuído não é a mesma coisa que por exemplo, um objeto escrito em Java. Um objeto distribuído tem uma interface exportada, a sua implementação pode ser feita em qualquer linguagem de programação (inclusive linguagens não orientadas a objetos) e pode executar em qualquer processador (Bosio, 2000). Os objetos distribuídos são independentes de linguagem de programação, *hardware*, sistema operacional, rede, protocolo de comunicação, espaço de endereçamento, compiladores, etc. Podem até mesmo ser chamados de componentes (Mainetti, 1997).

As chamadas aos objetos distribuídos são feitas após um mapeamento da IDL para linguagem de programação utilizada pelo objeto. Este mapeamento é feito automaticamente pelo pré-processador IDL e os módulos gerados são ligados ao código deste objeto. Isso permite que o objeto cliente e o objeto servidor possam ser desenvolvidos em linguagens de programação diferentes, o que acarreta uma independência entre os objetos.

### 2.2.3 ARQUITETURA

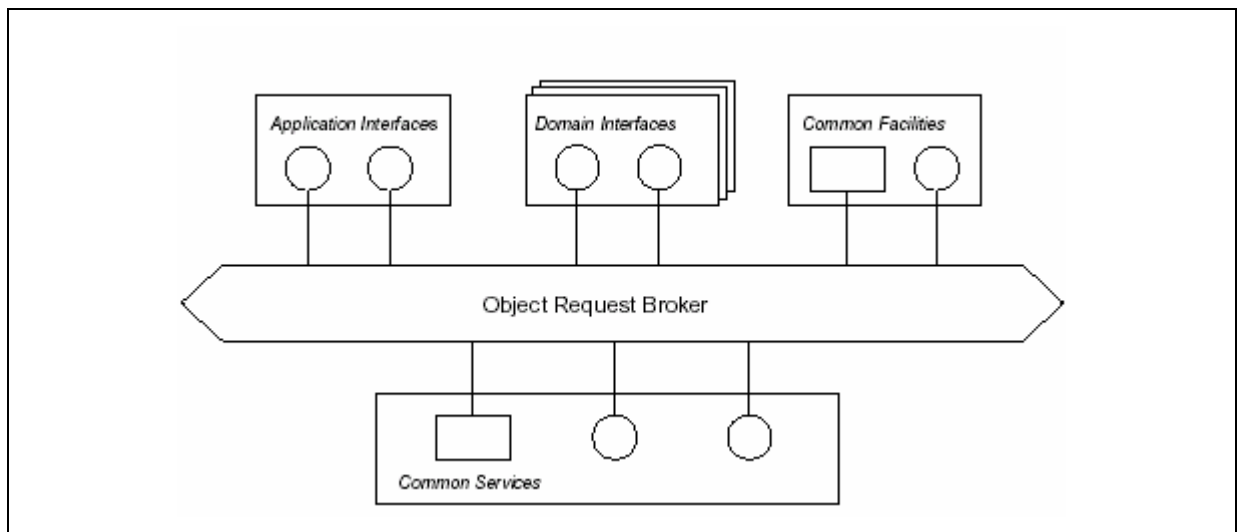
De acordo com Brose (2001), os componentes da arquitetura CORBA são:

- a) *Object Request Broker* (ORB): fornece um mecanismo para que as requisições dos clientes localizem o objeto servidor de uma forma transparente, constituindo o cerne da arquitetura CORBA;
- b) *Common Services*: objetos da infra-estrutura que estendem o ORB, oferecendo uma coleção de serviços necessários para construção de um aplicação distribuída;
- c) *Common Facilities*: inserem funcionalidades comuns utilizadas diretamente por objetos de negócio;

- d) *Domain Interfaces*: são as interfaces específicas dos domínios como: finanças, saúde, manufatura, comércio eletrônico, entre outros;
- e) *Application Interface*: são os objetos de negócio e aplicação, consistindo na camada mais alta da Arquitetura.

Esses componentes podem ser observados na Figura 2.

FIGURA 2 ESTRUTURA CORBA



Fonte: Silva (2000)

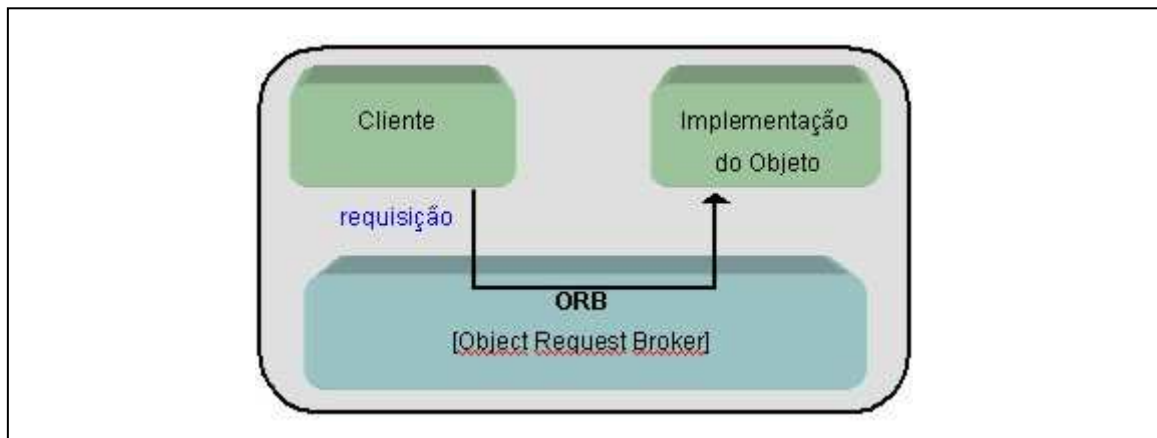
### **2.2.3.1 OBJECT REQUEST BROKER (ORB)**

O ORB é o mecanismo que provê serviços básicos para que um objeto distribuído possa interagir com outros. Ele intercepta a chamada de um objeto para outro e fica responsável em encontrar um objeto que atenda as necessidades do pedido. Encontrando o objeto chamado, o ORB passa os parâmetros para o mesmo, invoca os métodos necessários dele e retorna para o objeto que solicitou os resultados de todo esse procedimento. Dessa maneira, o usuário não precisa se preocupar onde tal objeto está localizado, em que sistema operacional ele roda ou qual programa foi usado para desenvolvê-lo.

Como a comunicação é feita de forma transparente, entre o cliente e a implementação do objeto desejado, o ORB é a principal arma na simplificação da programação em ambientes distribuídos, já que o mesmo libera os programadores de realizarem o árduo trabalho de programar a baixo nível os detalhes das invocações dos métodos. O ORB tem como principal

função, achar a implementação do objeto que atenda as necessidades da invocação de um determinado cliente. Após isso, fica responsável pela ativação (caso não esteja ativado) do mesmo, passar as informações necessárias para o objeto e devolver (quando for necessário) para o cliente o resultado desta operação. Todos esses passos são feitos de forma transparente para o cliente (Jacobsen, 2000). A Figura 3 mostra uma requisição oriunda de um cliente sendo enviada através do ORB a uma implementação de objeto.

FIGURA 3 UMA REQUISIÇÃO ENVIADA ATRAVÉS DO ORB.

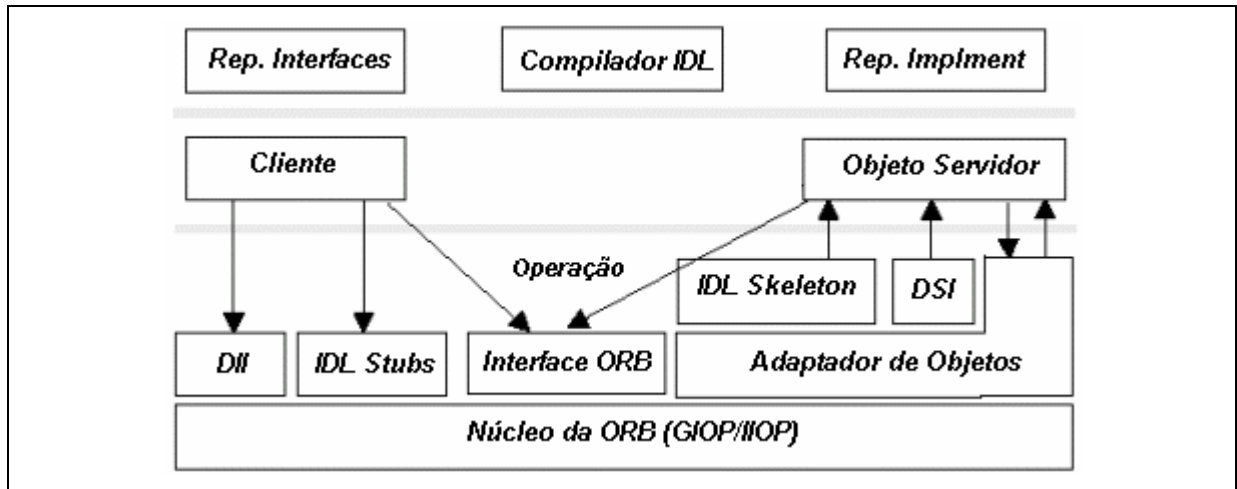


FONTE: Brose (2001).

### 2.2.3.1.1 COMPONENTES DO ORB

De acordo com Lima (2001), um objeto CORBA pode agir simultaneamente como cliente e servidor. Na Figura 4 é explicitada a estrutura de um ORB e suas interfaces, através das quais acontece a interação com os clientes e com as implementações de objetos.

FIGURA 4 ESTRUTURA DE UM ORB



Fonte: baseado em Silva (2000).

Os componentes da Figura 4, são explicados a seguir.

### 2.2.3.1.2 LADO CLIENTE

De acordo com Lima (2001), os clientes fazem requisições tendo acesso a uma referência de *interface* de um objeto e conhecendo o tipo do objeto e a operação desejada a ser executada. O cliente inicia uma requisição chamando rotinas do *stub* que são específicas do objeto ou construindo a requisição dinamicamente. Os componentes que compõem um objeto cliente são descritos a seguir:

- a) *stub*: são interfaces estáticas que definem como os clientes invocam métodos nos servidores, sem preocupações com detalhes de comunicação;
- b) *Dynamic Interface Invocation* (DII): permite a descoberta em tempo de execução dos métodos a serem invocados.
- c) *Interface Repository* (IR): fornece objetos persistentes que representam a informação da IDL disponível em tempo de execução. Permite obter e modificar descrições de todas interfaces de componentes registrados, os métodos oferecidos, assim como sua assinatura;
- d) interface ORB: consiste em *Application Program Interfaces* (API) para serviços locais que podem interessar a aplicação, sendo as mesmas para todos os ORBs, não dependendo da interface do objeto ou do adaptador de objetos.

### 2.2.3.1.3 LADO SERVIDOR

De acordo com Lima (2001), o ORB localiza o código da implementação do objeto apropriado e transfere o controle para a implementação do objeto através de um *skeleton* IDL ou de um *skeleton* dinâmico. O lado servidor não percebe a diferença entre uma invocação estática e uma dinâmica de um cliente.

Os componentes que compõe um objeto servidor são descritos a seguir:

- a) implementação do objeto: contém a implementação da funcionalidade do servidor. Pode conter a instanciação de diversos objetos no seu interior. Esta implementação pode ser codificada em linguagens como, por exemplo, Delphi, Java, C, C++, Ada e Smaltalk (Jacobsen, 2000);
- b) *skeleton*: fornece uma interface estática para cada serviço exportado pelo servidor. São específicos para a interface e o adaptador de objetos;
- c) *Dynamic Skeleton Interface* (DSI): permite tratamento dinâmico para invocações de objeto. Ao invés de ser acessada através de um *skeleton* específico de uma interface particular, a implementação do objeto é alcançada através de uma interface que provê acesso ao nome e parâmetros de um operação de maneira análoga ao DII. *Skeleton* dinâmicos são muito úteis para implementar pontes genéricas entre ORBs;
- d) adaptador de objeto: localizado sobre o núcleo dos serviços de comunicação do ORB, aceita requisições de serviços em nome dos objetos servidores. O adaptador de objeto oferece a geração e interpretação de referência de objetos e a ativação e desativação da implementação dos objetos (Bosio, 2000);
- e) repositório de implementações: fornece um repositório *run-time* de informações que permitem com que o ORB localize e ative implementações de objetos seguindo políticas relacionadas à ativação e execução de suas implementações;
- f) interface ORB: possui a mesma funcionalidade do objeto cliente.

De forma resumida, Lima (2000) afirma que o núcleo do ORB assegura funções de comunicação entre os objetos; os adaptadores gerenciam o direcionamento das comunicações ativando os objetos servidores entre outras coisas; os *stubs* e *skeletons* fornecem funções de adaptação entre a comunicação e o mecanismo de invocação de operações nos objetos servidores.



### 2.2.3.2 OBJECT SERVICES

Os *Object Services* (serviços do objeto) ajudam no gerenciamento e na manutenção dos objetos. Pode-se usar as interfaces disponibilizadas por eles para criar, manipular a segurança e determinar as suas respectivas localizações. Os serviços CORBA são (Lima, 2001):

- a) serviço de coleção: fornece interfaces CORBA para criar e gerenciar coleções genericamente;
- b) serviço de relacionamento: fornece um meio para criar associações dinâmicas (ou *links*) entre componentes que não se conhecem. Também fornece mecanismos para percorrer os links que agrupam estes componentes. Pode ser usado para forçar restrições de integridade referencial, checar relacionamentos de conteúdo e para qualquer tipo de ligação entre os componentes;
- c) serviço de controle de concorrência: fornece um gerente de *locks* que podem obter *locks* em nome de transações ou *threads*;
- d) serviço de segurança: fornece uma estrutura completa para segurança de objetos distribuídos. Suporta autenticação, listas de controle de acesso, confidencialidade e não repudição. Também gerencia a delegação de credencias entre objetos;
- e) serviço de tempo: fornece uma interface para sincronização de tempo em um ambiente de objetos distribuídos. Também fornece operações para definir e gerenciar eventos disparados por tempo.
- f) serviço de licenciamento: fornece operações para medir o uso de componentes e para assegurar uma compensação justa pelo seu uso. Suporta qualquer modelo de controle de utilização em qualquer ponto no ciclo de vida do componente;
- g) serviço de externalização: fornece um meio padrão para enviar e receber dados de um componente utilizando um mecanismo de *stream*;
- h) serviço ciclo de vida: define as operações para criar, copiar, mover e remover componentes da plataforma;
- i) serviço de transações: fornece coordenação “*two-phase commit*” entre componentes recuperáveis utilizando transações planas ou aninhadas;
- j) serviço de propriedades: fornece operações que permitem a associação de propriedades (nome-valor) a qualquer componente. Permite a associação dinâmica de propriedades ao estado do componente;

- k) serviço de nomes: permite que componentes localizem outros através de seu nome. Permite que objetos sejam registrados em diretórios de rede existentes, ou contextos de nomes (ISO-X.500, OSF-DCE, Sun-NIS+, Novel-NDS, Internet-LDAP);
- l) serviço de consulta: fornece operações de consulta aos objetos. É um superconjunto da *Structured Query Language* (SQL);
- m) serviço de persistência: fornece uma interface única para armazenar componentes em vários servidores de armazenamento (Base de Dados Objetos (ODBMS), Base de Dados Relacionais (RDBMS), e arquivos simples);

Os serviços de objetos do CORBA permitem que o desenvolvedor concentre seus esforços em seus objetos, sem ter que se preocupar com serviços a nível de sistema.

Pode-se desenvolver uma classe para um objeto e então usar os serviços para adicionar a funcionalidade que eles oferecem.

### **2.2.3.3 COMMON FACILITIES**

As facilidades de objetos definem um conjunto de serviços que podem ser utilizados por várias aplicações, mas que não são considerados fundamentais (Silva, 2000). Estes serviços estão divididos em grandes categorias (Brose, 2001):

- a) interfaces de aplicação: consistem de facilidades não padronizadas específicas para aplicações individuais, como um sistema de reservas ou sistema de gerenciamento de inventário;
- b) interfaces de domínio: consistem em facilidades para usuários finais em domínios de aplicações específicas. Neste caso, um domínio refere-se a um mercado vertical específico ou uma indústria.

### **2.2.3.4 APPLICATION OBJECTS**

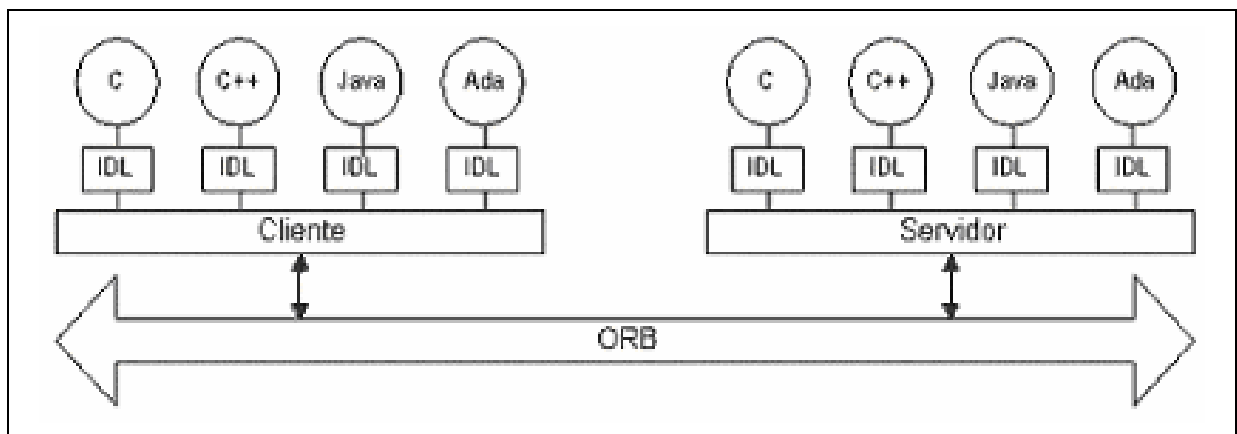
Os atuais objetos que formam o centro de uma aplicação compatível com o CORBA são chamados de *Application Objects* (objetos de negócio). Um objeto de negócio é uma maneira de descrever conceitos que são independentes de uma aplicação, como um cliente ou um pagamento. Uma aplicação é composta por um coleção de objetos de negócio.

Geralmente um objeto pode manipular somente uma única tarefa, enquanto que coleções de objetos de negócio podem ser usadas para manipular todo um processo de negócio. Como exemplo pode citar um gerenciamento de reservas para uma linha aérea. Para gerenciar a complexidade de uma aplicação de objetos de negócio distribuídos, os objetos de negócio escondem a complexidade do processamento de fundo (*back-end*) concentrando-se nas interfaces entre eles e as implementações.

## 2.2.4 IDL E INTERFACES

A *Interface Definition Language* (IDL) é a linguagem utilizada para descrever a estrutura de acesso a um objeto distribuído. Uma definição de interface escrita em IDL define completamente a interface e especifica totalmente cada parâmetro das operações, disponibilizando toda a informação necessária para desenvolver clientes que utilizem as operações desta interface (Brose, 2001). Através de um arquivo de definição de interface de uma implementação de objeto, de extensão IDL, o cliente possui toda a informação que necessita para obter uma determinada operação deste objeto. Uma especificação IDL é independente do ORB e da linguagem utilizada na implementação do cliente e da implementação do objeto (Figura 5), assim como da máquina e do sistema operacional no qual estes se executam.

FIGURA 5 INDEPENDÊNCIA DE LINGUAGEM DE PROGRAMAÇÃO



Fonte: Lima (2001)

As especificações IDL são formadas por definições de tipos, constantes, exceções, módulos e interfaces. Este conjunto de definições compõe um arquivo, que contém a

informação necessária para o interfaceamento entre o cliente e a implementação de objeto, tendo como elemento intermediário o ORB.

### 2.2.4.1 DECLARAÇÃO DE INTERFACE

Uma declaração de interface é composta pela palavra-chave *interface*, seguida de seu identificador e de declarações opcionais de herança, ou seja, interfaces previamente descritas cujas características (declarações) serão herdadas por esta interface. Em seguida vem seu corpo da interface delimitado por chaves e composto pelas já descritas declarações de tipos, constantes e exceções, acrescidas de declarações de atributos e operações. O Quadro 1, mostra um exemplo da declaração de uma interface.

QUADRO 1 DECLARAÇÃO DE INTERFACE

```
interface cliente{
    /* atributos */
    attribute long codigo;
    attribute string nome;
    /* operacoes */
    long operacao1(in string nome);
};

/* herança */
interface pessoaFisica:cliente{
    void operacao2(in long codigo, inout string nome);
};

interface Pagamento{
    attribute long qtDiasPagto;
};

/* herança múltipla */
interface PessoaFisicaEspecial:pessoaFisica,Pagamento {
    string getNome();
};
```

Uma interface descreve a lista de operações e atributos em uma implementação. São especificados em uma interface os tipos dos atributos e a sintaxe de chamada às operações de um objeto, inclusive os tipos de retorno e dos parâmetros, as exceções que podem ser retornadas e o contexto correspondente. Um objeto satisfaz uma interface caso ele possa ser o destinatário de todas as requisições descritas nela, inclusive aquelas referentes aos atributos.

Uma interface pode ser derivada de outra interface, denominada de interface base, através de uma declaração de herança, onde a interface derivada herda todos os elementos da interface base. O uso de mais de uma base por uma interface é chamado de herança múltipla.

Referências e elementos não podem ser ambíguos, o que pode ocorrer nos casos em que um mesmo identificador é utilizado em mais de uma interface base. Para resolver o problema da ambigüidade deve ser usado o operador de escopo “::” entre o nome da interface e o identificador do elemento referenciado.

Uma interface não pode ser especificada mais de uma vez como base direta de uma interface derivada. No entanto, são aceitas múltiplas ocorrências como bases indiretas.

### 2.2.4.2 OPERAÇÕES

Uma operação é uma entidade que denota um serviço que pode ser requisitado. Deve ser genérica, na medida em que é independente da implementação dos objetos e utiliza mecanismos de herança de interfaces da *IDL*. Uma operação possui uma assinatura, a qual descreve todos os valores de parâmetros e resultados possíveis, através da especificação dos parâmetros necessários para as chamadas de operações, dos resultados da operação e das exceções que podem ocorrer durante a execução do serviço. A sintaxe de uma operação é descrita a seguir (Quadro 2):

QUADRO 2 SINTAXE DE UMA OPERAÇÃO

<p><i>[oneway]</i>    &lt;op_type_spec&gt;    &lt;identifier&gt;    (param1,...,paramL)    <i>[raises</i>  <i>(except1,...,exceptN)] [context(name1,...,nameM)]</i></p>
---

Fonte: Siegel (1996)

onde:

- a) *oneway*: é uma declaração opcional que indica a semântica *best-effort* de execução, na qual a operação não retorna nenhum resultado, e não há sincronização entre o requisitante e o término da operação. O padrão é a semântica *at-most-once*, na qual garante que se uma operação retorna com sucesso, esta foi executada exatamente uma vez. No caso de exceção, foi executada uma vez no máximo. A semântica de execução é associada a cada operação, o que garante que tanto o cliente quanto a implementação do objeto sempre assumirão a mesma semântica;
- b) *op\_type\_spec*: é o tipo do retorno da operação, que é um parâmetro de saída distinto;
- c) *identifier*: é o nome da operação;
- d) parâmetros: define a assinatura da operação e podem possuir os modificadores para indicar a direção da informação: *in* significa que a informação passa do cliente para o servidor, *out* do servidor para o cliente e *inout* em ambas as direções;
- e) *raises* é uma expressão opcional seguida pela lista de exceções definidas pelo usuário (as exceções padrão são incluídas implicitamente) que podem ser sinalizadas para terminar o pedido, indicando que a operação não foi executada com sucesso. Tais exceções podem ser descritas por um registro, no qual são acompanhadas por informação adicional específica da exceção;
- f) *context*: é uma expressão opcional que fornece informações adicionais sobre a operação requerida e pode afetar a maneira com que esta operação é processado pela implementação do ORB.

## 2.2.5 A INTEROPERABILIDADE ENTRE ORBS

Um dos objetivos da arquitetura proposta pela OMG é permitir a interoperabilidade entre sistemas de objetos, independentemente das técnicas e tecnologias utilizadas nas implementações destes. Pelo fato de existirem muitos sistemas com características diferentes, torna-se necessário que ORBs diferentes possam trabalhar conjuntamente, de forma a permitir o acesso a implementações a partir dos mais diversos sistemas. Para que múltiplos ORBs

possam se comunicar, o núcleo ORB deve propiciar um serviço de comunicação sem limitações impostas por protocolos incompatíveis. Devido ao fato de nenhum dos protocolos para conexão de redes de computadores existentes satisfazer por completo as necessidades de todos os usuários, não existe um protocolo aceito como padrão (Silva, 2000).

Para que a interoperabilidade seja possível, a linguagem IDL, mantém-se inalterada independentemente do ORB utilizado. Com isso, é possível que uma requisição passe por múltiplos ORBs sem alterações em sua semântica, tornando a invocação de operações transparente para clientes e implementações de objetos.

Em uma máquina onde atuam mais de um ORB, um cliente pode acessar objetos implementados em qualquer um destes, podendo um objeto de um ORB passar como parâmetro uma referência de um outro objeto. Também serão acessíveis para este cliente os objetos que estiverem em uma máquina na qual existir um dos ORB aos quais ele tem acesso.

Nos casos em que for necessário acessar uma implementação de objeto que se executa em uma máquina que não possua um destes ORB, será necessário utilizar uma técnica que permita que as referências de objetos e as requisições de um dos ORB ao qual o cliente tem acesso, sejam transformadas para o formato aceito pelo ORB da implementação, tornando possível a sua identificação e a execução da operação solicitada (Silva, 2000).

### **2.2.5.1 TÉCNICAS PARA CONEXÃO DE ORBS**

Existem várias técnicas para conectar ORBs, mas estas podem ser organizadas em categorias mais abrangentes que permitem uma visão melhor da solução a ser adotada. A primeira destas seria utilizar uma *reference embedding* (referência embutida), que faria com que um objeto em um ORB seja visto como um objeto em um segundo ORB. Através desta técnica, uma invocação no segundo ORB ativaria uma implementação cuja tarefa seria fazer uma invocação no objeto do primeiro ORB, utilizando os dados embutidos na sua referência (Silva, 2000).

Uma outra técnica, adequada quando ORBs diferem em detalhes de implementações, mas possuem funcionalidades semelhantes, baseia-se em traduzir requisições da representação utilizada em um ORB para outro. Tal técnica implicaria na construção de *gateways* que

realizariam o processo de tradução entre os formatos utilizados por ORBs diferentes (Silva, 2000).

A outra alternativa seria fazer com que o mesmo objeto esteja disponível em mais de um ORB, desde que estes utilizem um mesmo adaptador de objetos, de cuja interface a implementação é dependente. Com isso, a interface do objeto poderia definir operações que permitam que o cliente obtenha a referência de um objeto equivalente em um outro ORB (Silva, 2000).



## 3 DESIGN PATTERNS

### 3.1 INTRODUÇÃO

Os *Design Patterns*, também conhecidos como padrões de projeto, têm origem no conceito de reuso de software. Existem problemas de análise e de projeto de sistemas que, embora estejam situados em contextos diferentes, podem ser solucionados de maneiras semelhantes. Dessa forma, analistas e projetistas podem descobrir como resolver um problema a partir de um trabalho antigo. Essas soluções podem ser transformadas num padrão para resolução desses problemas semelhantes.

Um padrão pode ser definido como sendo um esboço, em vez da implementação específica, ou seja, um modelo a ser seguido durante a construção do software (Coad, 1995). Pode-se ainda definir um padrão como sendo uma solução recorrente para problemas comuns no desenvolvimento de software. Cada desenvolvedor pode ter seus próprios padrões e compartilhar padrões com outros desenvolvedores.

Segundo Gamma (1995), os *patterns* são definidos como sendo a descrição de um problema que ocorre muitas e muitas vezes e portanto eles descrevem o coração da resolução do problema, de forma que sua solução possa ser usada por milhares de vezes, sem ter que se percorrer, novamente, o mesmo caminho.

Segundo Mowbray (1997), quando padrões são relacionados em conjunto, eles formam uma linguagem que provê processos metódicos para a resolução de problemas de desenvolvimento de software. Ambos, padrões e linguagem de padrões, ajudam os desenvolvedores a compartilhar conhecimentos adquiridos com a experiência de cada um, ajudam também a aprender novos estilos de arquiteturas, novas maneiras de desenvolver projetos, e ajudam ainda os desenvolvedores novatos a desviarem de falhas e erros já cometidos por outros mais experientes.

Em geral, *patterns* ajudam a reduzir a complexidade em muitas situações da vida real. Por exemplo, em algumas situações onde a seqüência de ações é crucial na ordem de execução de uma determinada tarefa. Ao invés de escolher de quase um número de combinações possíveis de ações, os *patterns* permitem a solução de problemas fornecendo um conjunto de combinações já previamente testadas (Pree, 1995).

Em termos de orientação a objetos, *Design Patterns* identificam classes, instâncias, seus papéis, colaborações e a distribuição de responsabilidades. Seriam, então, descrições de classes e objetos que se comunicam, que são implementados a fim de solucionar um problema comum em um contexto específico. *Design Patterns* tem vários usos no processo de desenvolvimento de software orientado a objetos (Schneide, 1999):

- a) formam um vocabulário comum que permite uma melhor comunicação entre os desenvolvedores, uma documentação mais completa e uma melhor exploração das alternativas de projeto;
- b) reduz a complexidade do sistema através da definição de abstrações que estão acima das classes e instâncias. Um bom conjunto de padrões aumenta muito a qualidade do programa;
- c) constituem uma base de experiências reutilizáveis para a construção de software;
- d) funcionam como peças na construção de projetos de software mais complexos;
- e) podem ser considerados como micro-arquiteturas que contribuem para a arquitetura geral do sistema;
- f) reduzem o tempo de aprendizado de uma determinada biblioteca de classes. Isto é fundamental para o aprendizado dos desenvolvedores novatos;
- g) quanto mais cedo são usados, menor será o retrabalho em etapas mais avançadas do projeto.

De acordo com Gamma (1995), *Design Patterns* aumentam a expressividade e o nível de descrição suportada pela orientação a objetos. Inversamente, estes conceitos podem ser aplicados à:

- a) encapsulamento e abstração: cada *Design Pattern* engloba um problema em um determinado contexto e a sua solução, com limites bem definidos entre o espaço do problema e da solução. Também corresponde a uma abstração que abrange conhecimento e experiências sobre o domínio da aplicação;
- b) extensibilidade e adaptabilidade: cada *Design Pattern* deve ser extensível para que outros possam moldá-lo a fim de resolver um problema mais amplo. Deve ser também adaptável para que sua solução possa servir em uma vasta gama de implementações;
- c) geração e composição: cada padrão, uma vez aplicado, gera como consequência um contexto inicial de um ou mais *Design Patterns*. Estes poderão ser aplicados em

- conjunto, com o objetivo de alcançar uma solução global e completa;
- d) equilíbrio: os padrões devem procurar atingir um equilíbrio entre suas forças e restrições. Deve-se minimizar os conflitos dentro do espaço de solução do problema, e dar um porquê para cada passo ou regra dentro do *Design Pattern*.

### 3.1.1 ORIGENS

A origem dos *Design Patterns* encontra-se em um trabalho feito por um arquiteto chamado Christopher Alexander durante os anos 70 com seu livro "*A Pattern Language*". Este arquiteto aplicou o conceito de *Design Patterns* a repetições de formas na arquitetura. Ele argumentava que os métodos de arquiteturas não atendiam às reais necessidades dos indivíduos e da sociedade. Ele queria criar estruturas que melhorassem a qualidade de vida das pessoas (Mowbray, 1997).

No final da década de 70 e início dos anos 80, Ward Cunningham e Kent Beck, perceberam que poderiam desenvolver um conjunto de padrões para serem aplicados no desenvolvimento de interfaces do usuário em Smalltalk. Esses padrões foram chamados de *Model-View-Controller* (MVC) (Pree, 1995). No mesmo período, Jim Coplien estava desenvolvendo um catálogo de padrões C++ chamado idiomas. Enquanto isso, Erich Gamma estava trabalhando em sua tese de doutorado sobre desenvolvimento de software orientado a objetos e reconheceu a importância de acumular explicitamente as estruturas de projetos que se repetiam com frequência (Schneide, 2001).

Essas e outras pessoas intensificaram suas discussões em uma série de *workshops* do *Object-Oriented Programming Systems Languages and Applications* (OOPSLA) organizado em 1991 por Bruce Anderson, e em 1993 a primeira versão de um catálogo de padrões estava esboçada (Pree, 1995).

Em 1994, este conceito tornou-se popular pela primeira vez com a publicação do livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" (Gamma, 1995). Seu trabalho focou exclusivamente no nível micro arquitetural, que identifica estruturas básicas para construção de outros componentes de software, que provê padrões com diversas utilidades (Mowbray, 1997).

Atualmente, padrões para o desenvolvimento de software é um dos assuntos mais emergentes da comunidade de orientação a objetos, resultando um grande número de publicações a seu respeito (Cunha, 2001).

### **3.1.2 ESTRUTURA DE UM *DESIGN PATTERN***

Quanto à estrutura formadora de um *Design Pattern*, existem vários autores que apresentam propostas diferentes. Segundo Mowbray (1997), um padrão de projeto possui quatro elementos essenciais:

- a) nome do padrão: tem o objetivo de identificar um padrão. Usa-se, geralmente, uma ou duas palavras.
- b) problema: indica quando aplicar o padrão e descreve o problema para o qual foi construído, explicando os seus detalhes.
- c) solução: diz respeito à solução encontrada para resolver o problema. Não se deve descrever um projeto ou implementação em particular, porque um padrão deve servir de modelo para outras situações um pouco diferentes. Juntamente com a solução, pode-se ter diagramas que auxiliam o seu entendimento. Os diagramas podem ser construídos com base em alguma metodologia, como por exemplo, a orientação a objetos.
- d) conseqüências: são os resultados da utilização do padrão, podendo-se dar a idéia de custo-benefício do seu uso.

Alguns autores incluem pequenos exemplos de código em alguma linguagem de programação orientada a objetos, o que é muito interessante para enriquecer a descrição de um padrão.

### **3.1.3 COMO UTILIZAR UM *DESIGN PATTERN***

De acordo com Gamma (1995) para utilizar um *pattern* orientado a objetos, de maneira conveniente, deve-se verificar os seguintes itens:

- a) estudar os *patterns* aos quais já se tenha acesso, dando uma atenção especial para as aplicações e conseqüências de sua utilização para garantir que este é o *pattern* ideal para a resolução do problema;

- b) estudar a estrutura, os participantes e as colaborações, para entender as classes e objetos no *Design Pattern* e como elas se relacionam;
- c) observar a seção com exemplo do código fonte do *Design Pattern*;
- d) escolher nomes significantes para os padrões participantes. Os nomes para estes participantes são geralmente muito abstratos para aparecerem diretamente na aplicação. Contudo, é muito útil incorporar o nome do participante no nome que aparece na aplicação, pois isto ajuda a fazer com que o padrão fique mais explícito na implementação;
- e) definir as classes. Declarar suas interfaces, estabelecer suas relações de herança, e definir as instâncias das variáveis que representam dados e referências aos objetos. Identificar classes existentes na aplicação que serão afetadas pelo *Design Pattern* e as modificar de acordo com a necessidade;
- f) definir nomes de aplicações específicas para operações. Os nomes geralmente dependem da aplicação. Utilizar as responsabilidades e colaborações associadas a cada operação como guia, sendo consistente quanto à sua convenção de nomenclatura;
- g) implementar as operações para cuidarem das responsabilidades e colaborações. A seção de implementação oferece dicas para implementação, que podem ser úteis.

Nenhuma discussão de como usar *Design Patterns* seria completa se não houvesse pelo menos algumas poucas palavras sobre como não usá-los. Eles não devem ser usados indiscriminadamente. Da mesma forma que proporcionam flexibilidade e variabilidade que podem complicar um projeto e reduzir sua performance (Gamma, 1995). Um *Design Pattern* só deve ser aplicado quando sua flexibilidade valer o custo necessário. A seção de conseqüências ajuda bastante na hora de tomar esta decisão.

### **3.1.4 COMO SELECIONAR UM *DESIGN PATTERN***

Com um catálogo abrangente para *Design Patterns* pode ser difícil encontrar aquele que endereça um problema particular, especialmente se o catálogo é novo e se tem pouca familiaridade com ele. A seguir são descritas algumas abordagens para encontrar um *Design Pattern* para resolver um problema (Gamma, 1995):

- a) considerar como os *patterns* resolvem os problemas. Como os *patterns* ajudam a achar objetos apropriados, a granularidade dos objetos, especificação das interfaces, e vários outros métodos com os quais eles resolvem os problemas de projeto;
- b) ver qual o objetivo de cada *pattern*. Isto, ajuda na seleção do padrão mais apropriado;
- c) examinar como os *Design Pattern* se relacionam;
- d) examinar as causas de reprojeto. Estudar as principais causas pelas quais frequentemente projetos são reestruturados e procurar pelo *pattern* que possa evitar este problema;
- e) considerar o que pode ser variável na aplicação. Deve-se considerar o que pode sofrer variações sem ter de começar um novo projeto.

### 3.1.5 COMO DESCREVER UM *DESIGN PATTERN*

Cada *pattern* é um conjunto de partes, que expressa uma relação entre um certo contexto, um certo sistema de forças que ocorre repetidamente neste contexto e certa configuração de software que permite que estas forças se resolvam. Para descrever um *pattern* as notações gráficas, embora importantes são insuficientes. A utilização de uma estrutura facilita o aprendizado, a comparação e a sua utilização. Esta estrutura, de acordo com Mowbray (1997), é composta das seguintes partes:

- a) nível da aplicabilidade: identifica em qual nível do modelo de escalabilidade o padrão pode ser utilizado, ou seja, o modelo será aplicado às classes, à aplicação, ao sistema, à infraestrutura ou utilizado de forma global;
- b) tipo da solução: identifica o tipo da solução proposta. Poderá ser de quatro tipos:
  - *software*: quando o padrão envolver a criação de uma novo *software*;
  - tecnológico: quando o padrão é utilizado em problemas relacionados à adoção de uma tecnologia, por exemplo, Java, Delphi, C++, etc.;
  - processo: definem padrões relacionados ao processo de criação de uma aplicação;
  - papel: quando o padrão pode ser aplicado para alocar responsabilidades para um conjunto de classes.
- c) nome do *Design Pattern*: identifica o padrão;

- d) problema: identifica claramente o problema, que o *Design Pattern* pode ajudar a resolver;
- e) diagrama: expressa uma notação gráfica para esclarecer a solução proposta.
- f) forças primordiais: identifica sobre qual força primordial o padrão pode ser aplicado. Estas forças podem ser : gerência de funcionalidades, gerência de performance, gerência de complexidade, gerência de alteração, gerência de tecnologia de informação ou tecnologia de transferência;
- g) aplicabilidade: identifica quais são os motivos para utilização do *Design Pattern*;
- h) resumo da solução: esta seção, explica de maneira resumida como um determinado problema é resolvido pelo padrão;
- i) benefícios: identifica quais serão os benefícios após a adoção do padrão;
- j) conseqüências: identifica as conseqüências da sua utilização;
- k) variantes da solução: muitos padrões frequentemente possuem variações como alternativas de construção. Esta seção contempla esse tipo de extensão no *Design Pattern*;
- l) reescalando em outros níveis: esta seção descreve a relevância do padrão em outros níveis de escalabilidade;
- m) soluções relacionadas: descreve a relação entre os *patterns*. Identifica quais as diferenças entre eles e com quais deles podem ser usados;
- n) relato da aplicabilidade do padrão: esta seção tem o objetivo de descrever como ocorre o problema ou geralmente mostra informações sobre a experiência de sua utilização, que seja útil ou interessante.

## 3.2 DESCRIÇÃO DOS DESIGN PATTERNS

A seguir serão descritos alguns *Design Patterns* aplicados a sistemas distribuídos, mais especificamente para a arquitetura CORBA utilizando o *middleware* Visibroker. Os *Design Patterns* aqui descritos serão implementados neste trabalho.

### 3.2.1.1 DISTRIBUTED CALLBACK

Este *Design Pattern* aplicado a sistemas distribuídos, descreve uma técnica para maximizar o paralelismo entre os objetos. Este padrão, proposto por Mowbray (1997), tem o

objetivo de transformar operações seqüenciais em operações paralelas, o que reduz o tempo de espera de uma operação requisitada por um objeto cliente. O *pattern* é descrito a seguir (Mowbray, 1997):

- a) nível da aplicabilidade: aplicação;
- b) tipo da solução: *software*;
- c) nome do *Design Pattern*: *Distributed Callback*;
- d) problema: um objeto cliente necessita de um serviço, mas não pode ficar esperando até que o objeto servidor possa responder ao serviço requisitado;
- e) forças primordiais: gerência de performance;
- f) aplicabilidade: pode ser aplicado quando:
  - o objeto cliente necessita gerenciar uma ou mais atividades concorrentemente;
  - a operação que foi requisitada pelo objeto cliente precisa de um tempo considerável para ser executado e o objeto cliente não pode esperar este tempo para continuar o processamento;
  - o objeto cliente possui muitas operações a serem requisitadas e pode receber o resultado destas operações de uma forma não seqüencial;
  - o objeto cliente necessita continuar processando instruções locais enquanto são enviadas requisições CORBA;
  - o objeto cliente só precisa aceitar os resultados, disponíveis em intervalos de tempo específicos.
- g) resumo da solução: primeiramente são definidas as interfaces para o cliente e o servidor. Para cada método que o cliente pretender aplicar o padrão, o cliente deve definir um objeto *callback* utilizando a IDL. Este objeto contém todos os parâmetros para o retorno das informações requeridas do servidor. Para aumentar a performance, o cliente *callback* deve definir uma operação *oneway* do CORBA. Cada operação do servidor é convertida de síncrona para assíncrona com a operação *oneway* com os parâmetros adicionais para a interface *callback* (Quadro 3). Para esta conversão deve-se seguir os seguintes critérios:
  - os parâmetros permanecem iguais;
  - os parâmetros inout devem ser convertidos em parâmetros de entrada e adicionados à lista de informação de retorno do callback cliente;



- os parâmetros out devem ser removidos da IDL e adicionado à lista de parâmetros de retorno do *callback* cliente;
  - finalmente, se a operação tem um tipo de retorno, deve ser substituído com um retorno void e adicionado como um parâmetro na interface do *callback* cliente.
- h) benefícios: o cliente não necessita esperar pelo servidor para completar a operação;
- i) conseqüências: as exceções ficam menos úteis com chamadas *oneway*, fica mais difícil averiguar no caso de erros ou fracassos e conferir os resultados parciais ou o estado do processo;
- j) variantes da solução: uma adição útil para a solução é uma assinatura adicional para o cliente e implementação de objeto trocar informação de progresso. A operação de progresso pode ser implementada no cliente (*pull-mode*) ou na implementação do objeto (*push-mode*).
- k) reescalando em outros níveis: As conseqüências de usar este padrão fica mais complexa em níveis mais altos da aplicação. Em particular, a falta de uma resposta garantida se torna uma preocupação mais séria. Também, a inabilidade para usar uma exceção para carregar informação de erro impede o uso deste padrão no sistema em níveis globais;
- l) soluções relacionadas: este padrão pode ser utilizado de maneira comunitária com o padrão de notificação.

### QUADRO 3 INTERFACE *CALLBACK*

```

/* operação normal (sincrona) */
...
string opl(in string nome);
...

/* utilizando o padrão callback */

module DC
{
  /* interface implementada pelo cliente */

  interface foo_callback
  {
    oneway void opl_cb(in string retorno, in long status);
  };
};

```

```

/* esta interface é implementada pelo objeto servidor;          */
/* é adicionado como parâmetro o objeto cliente para que possa */
/* enviar a resposta após o método ser executado                */

interface foo
{
    oneway void op1(in string nome, in foo_callback client);
};
};

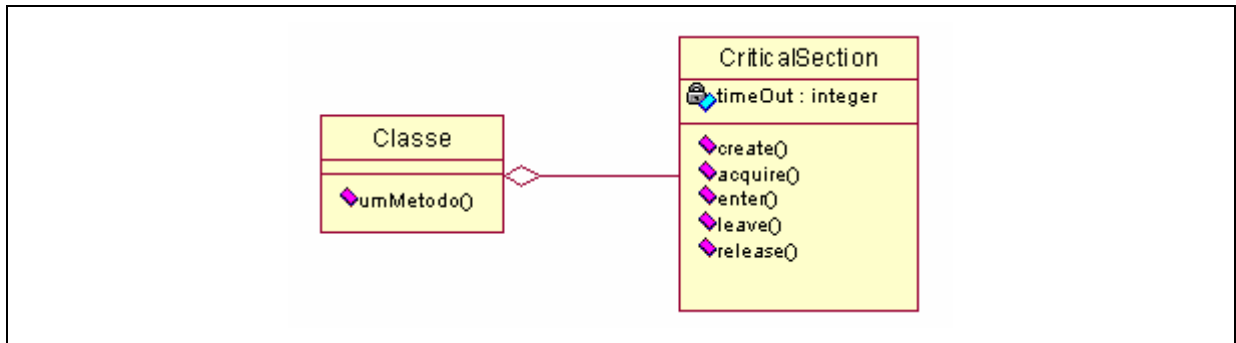
```

Fonte: Mowbray (1997).

### 3.2.1.2 LOCK

Este padrão também é proposto por Mowbray (1997) e utiliza a seguinte descrição:

- a) nível da aplicabilidade: aplicação;
- b) tipo da solução: tecnológica;
- c) nome do *Design Pattern*: *Lock*;
- d) problema: como gerenciar o acesso concorrente aos objetos distribuídos, para evitar problemas como dois objetos acessando uma sessão crítica e *deadlock*;
- e) forças primordiais: gerenciamento da funcionalidade;
- f) aplicabilidade: pode ser aplicado quando em um ambiente distribuído é possível um ou mais clientes acessarem um objeto simultaneamente. Se essa operação só pode ser acessada por um objeto para preservar sua integridade, este *Design Pattern* pode ser utilizado;
- g) resumo da solução: para coordenar o acesso a uma implementação de um objeto compartilhado, é definida agregação entre este objeto é que define uma sessão crítica (Figura 6). Então no início do método onde deverá ser feito o controle de acesso, é setado uma sessão crítica e ao final esta sessão é liberada para o acesso de outros clientes;
- h) benefícios: permite que diferentes processos executem em paralelo, sem comprometer a integridade da aplicação;
- i) conseqüências: ao utilizar este padrão sobre métodos que demoram um tempo considerável para serem executados, a performance fica prejudicada;
- j) exemplo: um exemplo deste padrão implementado na linguagem de programação *object pascal* é descrito no Quadro 4.

FIGURA 6 DIAGRAMA DE CLASSES DO PADRÃO *LOCK*QUADRO 4 EXEMPLO DO PADRÃO *LOCK*

```

unit exemploLock_impl;

interface
uses
  ...
  SyncObjs; // unit do CriticalSection
  ...

Tteste = class(TinterfacedObject, persistentLayer_i.teste)
  Protected

  SessaoCritica : TCriticalSection; //controla sessão crítica

  Public
  constructor Create;
  function umMetodo ( const stm : AnsiString): AnsiString;
  ...
end;

implementation

constructor Tconexao.Create;
begin
  inherited;
  SessaoCritica := TCriticalSection.Create; //cria uma sessão crítica
end;

function Tteste.umMetodo ( const stm : AnsiString): AnsiString;
begin
  SessaoCritica.Acquire; //define uma sessão crítica
  Try
    // implementação do metodo
  finally
    SessaoCritica.Release; //libera a sessão crítica
  end;
end;
end;

```

```
initialization
end.
```

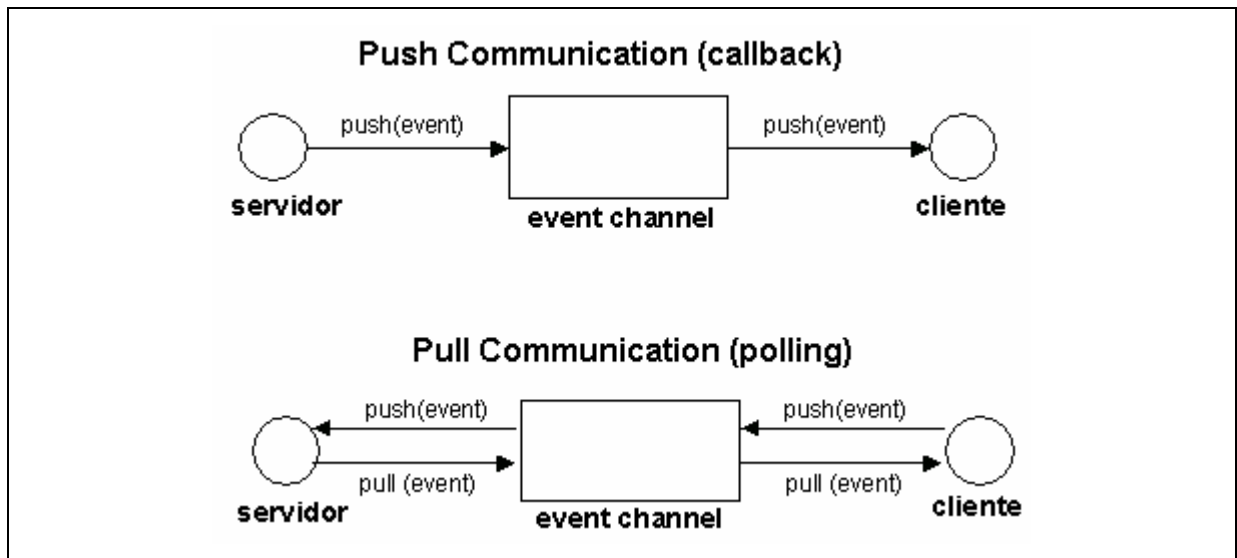
### 3.2.1.3 NOTIFICAÇÃO (EVENT SERVICE)

Este *Design Pattern* é descrito a seguir:

- a) nível da aplicabilidade: sistema;
- b) tipo da solução: tecnológica;
- c) nome do *Design Pattern*: notificação;
- d) problema: um objeto distribuído necessita enviar ou receber notificações que identificam o estado de outros objetos distribuídos que estão interagindo com ele;
- e) forças primordiais: gerenciamento de mudanças e gerenciamento da complexidade;
- f) aplicabilidade: este padrão pode ser aplicado quando:
  - é necessário ter um gerenciamento de eventos na aplicação;
  - é necessário informar a todos os objetos distribuídos sobre um evento ocorrido;
- g) resumo da solução: este padrão utiliza o serviço de eventos do CORBA e é utilizado para comunicação entre objetos em um ambiente distribuído. Os eventos são enviados ou recebidos através de um canal compartilhado (Figura 7). Existem dois modelos do serviço de eventos:
  - *push model*: ocorre quando um servidor transmite um evento para o cliente. Este modelo é similar ao que ocorre no *pattern Callback*;
  - *pull model*: o cliente explicitamente verifica se existe algum evento no canal de eventos (*event channel*);
- h) benefícios: suporta os dois modelos de notificação, possibilitando uma versatilidade na implementação. Os clientes podem enviar ou receber eventos sem a necessidade da definição de uma *interface* IDL para ele, embora possa ser associada;
- i) reescalando em outros níveis: este padrão pode ser utilizado para permitir a coordenação entre sistemas distribuídos, como por exemplo, avisando aos objetos quando são desabilitados ou quando um deles sofre alguma alteração que tenha influência no sistema;

- j) soluções relacionadas: este padrão pode ser associado ao padrão *Distributed Callback*, pois quando o servidor acaba a execução de uma tarefa requisitada pelo cliente pode ser enviado um evento informando que a operação foi executada;
- k) exemplo: no Quadro 5 mostra como é feita a implementação para ter acesso ao serviço de eventos do CORBA.

FIGURA 7 PATTERN DE NOTIFICAÇÃO



Fonte: Mowbay (1997).

QUADRO 5 EXEMPLO DO PATTERN NOTIFICAÇÃO

```

...
// cria e registra com BOA
PushSupplier_Skeleton:=TpushSupplierSkeleton.Create('Notificacao',
                                                    TPushSupplier.Create);
BOA.SetScope( RegistrationScope(1) );
BOA.ObjIsReady(PushSupplier_Skeleton as _Object);

//liga com o canal de eventos e pega o objeto Supplier Admin
Event_Channel := TeventChannelHelper.bind;
Supplier_Admin := Event_Channel.for_suppliers;

//pega o push cliente e registra o objeto servidor
Push_Consumer := Supplier_Admin.obtain_push_consumer;
Push_Consumer.connect_push_supplier(PushSupplier_Skeleton);
...

```

Para utilizar o padrão notificação, é necessário executar o serviço de eventos do CORBA. Para fazer isso o Visibroker possui a seguinte vista no Quadro 6:

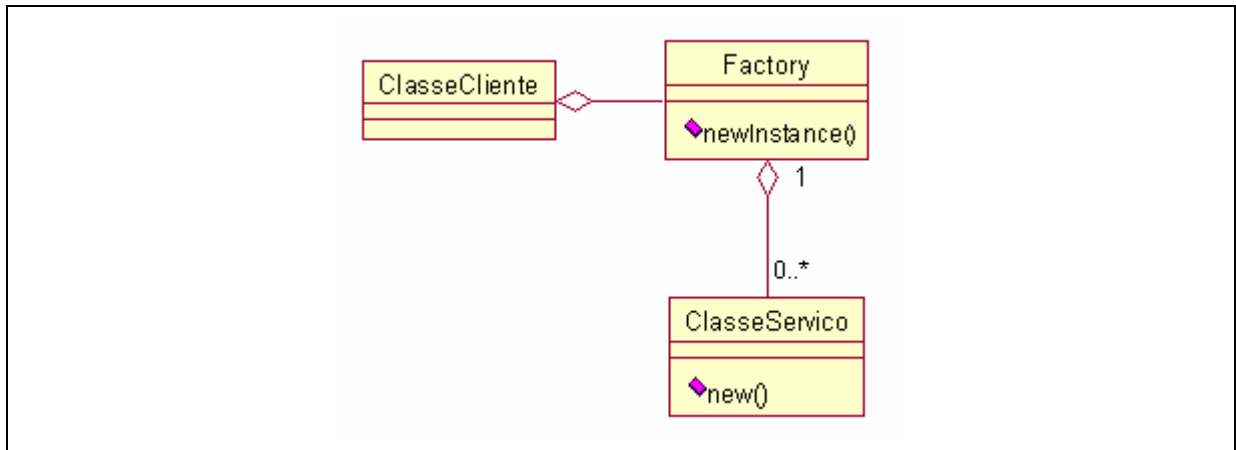
#### QUADRO 6 SINTAXE DO SERVIÇO DE EVENTOS

```
start vbj com.inprise.vbroker.CosEvent.EventServer -ior <Nome IOR>  
          <canal de eventos>
```

### 3.2.1.4 FACTORY

O *Design Pattern Factory* possui as seguintes características:

- a) nível da aplicabilidade: aplicação;
- b) tipo da solução: *software*;
- c) nome do *Design Pattern*: *Factory*;
- d) problemas: cada objeto distribuído necessita de uma nova implementação de um outro objeto distribuído a ele associado;
- e) forças primordiais: gerência de complexidade e gerência de performance;
- f) aplicabilidade: define uma *interface* para criar um objeto que deve possuir uma única instância para cada objeto distribuído que requisitar suas funcionalidades. Pode ser utilizado também quando é necessário delegar a responsabilidade de uma classe para uma ou diversas subclasses.
- g) resumo da solução: para aplicar o padrão *factory*, deve-se criar uma classe responsável pelas diversas instâncias do objeto que possuem a implementação. Quando um objeto cliente efetuar uma ligação com um objeto remoto, este instancia um novo objeto e devolve a referência para o cliente;
- h) benefícios: permite múltiplas instâncias para uma única interface; provê um mapeamento de uma instância do objeto;
- i) modelo: a Figura 8, apresenta o diagrama de classes do padrão *factory*, onde um objeto “classeCliente”, requisita uma nova instância para um objeto *factory* e este é responsável pela criação de um objeto “classeServico”. A referência do objeto criado é devolvido para o objeto “classeCliente”.

FIGURA 8 DIAGRAMA DE CLASSES DO PADRÃO *FACTORY*

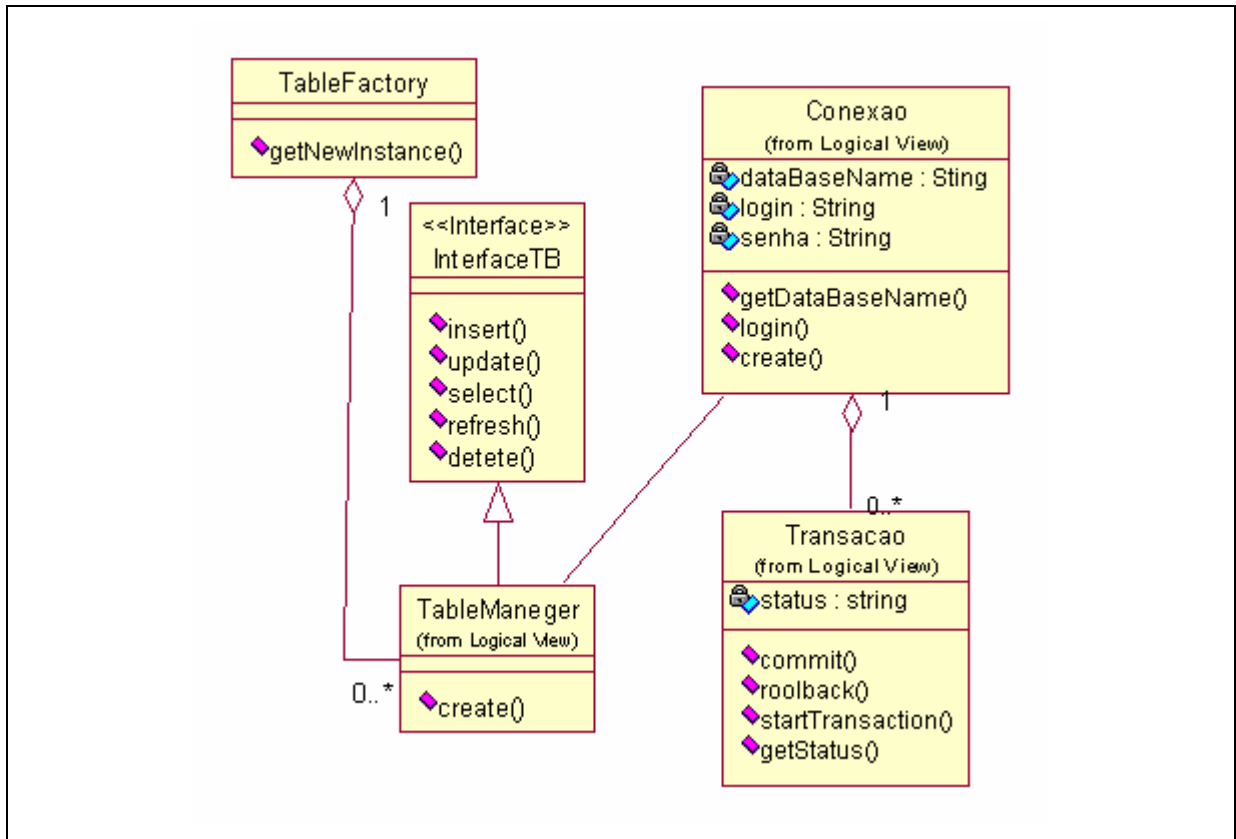
### 3.2.1.5 PERSISTENT LAYER

O *Design Pattern Persistent Layer* é descrito a seguir:

- a) nível da aplicabilidade: sistema;
- b) tipo da solução: *software*;
- c) nome do *Design Pattern*: *persistent layer*;
- d) problema: um objeto muitas vezes necessita salvar as informações referentes a seus atributos em um mecanismo persistente como, por exemplo, um banco de dados relacional. Então este objeto necessita de um mapeamento para manipular o acesso a este banco de dados;
- e) forças primordiais: gerência de alterações;
- f) aplicabilidade: é aplicado quando um objeto necessita persistir seus dados em um banco de dados relacional mas também pode ser utilizado como extração de dados de um banco de dados relacional;
- g) resumo da solução: primeiramente o objeto persistente deve implementar uma *interface* da classe *interfaceTB* (Figura 9). Após essa definição ele deve fazer uma ligação com um objeto que provê um mecanismo de acesso a dados, ou seja, deve estabelecer uma ligação com um objeto distribuído *tableFactory*, que utiliza o padrão *factory*, passando como parâmetro o usuário, a senha e o nome do driver de conexão com o banco de dados. Então se o usuário e senha tiver resultado positivo é criada uma nova instância do objeto *tableManager*, para que o objeto possa interagir com o banco de dados. Quando um objeto requisita a extração de

um conjunto de dados para um servidor de acesso a dados, esta informação é transferida através da linguagem XML, acarretando uma maior portabilidade entre os objetos distribuídos;

FIGURA 9 DIAGRAMA DE CLASSES DO *DESIGN PATTERN PERSISTEN LAYER*



- h) benefícios: permite a separação da lógica da aplicação com a lógica de manipulação de acesso ao banco de dados e provê um mecanismo padrão para persistência dos dados;
- i) conseqüências: como os dados são convertidos em XML para serem enviados a um objeto que utiliza o padrão *persistent layer*, isso acarreta um tempo maior de resposta do servidor de acesso a dados;
- j) variantes da solução: existem algumas implementações de CORBA que possuem um mecanismo de persistência, mas não é o caso do Visibroker, se este serviço existisse poderia complementar este padrão;
- k) exemplo: a IDL de um objeto que utiliza o padrão *persistent layer* pode ser observada no Quadro 7, que mostra a IDL utilizada neste trabalho;



- l) diagrama: o diagrama de classes do servidor de acesso a dados é mostrado na Figura 9.

#### QUADRO 7 IDL DO *DESIGN PATTERN PERSISTENT LAYER*

```

Typedef sequence<string> strArray;

Exception ERR_CONEXAO {
    string descricaoErro;
    long codigoErro;
};

exception ERR_DB {
    string descricaoErro;
    long codigoErro;
};

interface conexao
{
    attribute string dataBaseName;
    boolean login(in string driver, in string login, in string senha) raises
(ERR_CONEXAO);
    boolean desconecta();
    Object getConexao();
};

interface tableManager
{
    void setConexao(in conexao con);
    string select(in string stm);
    string execStorageProcedure(in string nome, in strArray parametros)
raises (ERR_DB);
    long insert(in strArray colName, in strArray colValue, in string
tableName) raises (ERR_DB);
    long insertDB(in string stm) raises (ERR_DB);
    long update(in strArray colName, in strArray colValue, in string
tableName, in strArray where) raises (ERR_DB);
    long updateDB(in string stm) raises (ERR_DB);
    long delete(in strArray where, in string tableName) raises (ERR_DB);
    long deleteDB(in string stm) raises (ERR_DB);
    string refresh(in string stm) raises (ERR_DB);
};

```

### 3.2.1.6 TEMPLATE

Este *Design Pattern* foi proposto por Gama (1995) e adaptado para este trabalho. Ele é descrito a seguir:

- a) nível da aplicabilidade: aplicação;
- b) tipo da solução: *software*;

- c) nome do *Design Pattern*: *template*;
- d) problema: muitas vezes um determinado processo, como por exemplo, a criação de uma interface com o usuário, segue um mesmo padrão, mudando muitas vezes os títulos da aplicação, os campos para visualizar as informações, entre outros. A criação de um processo muito parecido ou muito utilizado, quando feito de forma manual, torna-se um trabalho exaustivo e desta forma suscetível à falhas;
- e) forças primordiais: gerência da funcionalidade e gerência de complexidade;
- f) aplicabilidade: este padrão pode ser utilizado para definição das interfaces com o usuário visto que as interfaces são muito parecidas. No Quadro 8, é apresentado um exemplo de um *template* para criação de uma interface *WEB* com o usuário. Este exemplo mostra uma interface para ser utilizado na linguagem *object pascal*. Os pontos configuráveis são identificados com uma *tag*, que possui a seguinte sintaxe: *#nomeTag*;
- g) resumo da solução: quando for necessário criar uma interface com o usuário, simplesmente é configurado o *template* a ser utilizado e as *tags*, onde ele deve colocar as informações que são específicas de cada processo. Desta forma todas as *interfaces* são criadas de maneira padrão;
- h) benefícios: reduz o tempo de implementação de um novo processo, pois grande parte da implementação já estará pronta;
- i) variantes de solução: esta solução poderia ser utilizada para geração da *interface IDL* ou da implementação de um objeto distribuído;
- j) reescalando em outros níveis: a implementação do padrão *template* pode ser utilizada em todos os níveis da aplicação.

QUADRO 8 EXEMPLO DO *DESIGN PATTERN TEMPLATE*

```

object #classeForm: T#classeForm
  OldCreateOrder = False
  PageProducer = AdapterPageProducer
  Left = 297
  Top = 188
  Height = 150
  Width = 215
  object AdapterPageProducer: TadapterPageProducer
    HTMLDoc.Strings = (
      '<html>'
      '<head>'
      '</head>'
      '<body>'
      '<#STYLES><#WARNINGS><#SERVERSCRIPT>'
      '</body>'
      '</html>')
    Left = 48
    Top = 8
  object AdapterForm: TadapterForm
    object AdapterFieldGroup: TadapterFieldGroup
      Adapter = #classe.DataSetAdp
    End
    object AdapterCommandGroup: TAdapterCommandGroup
      DisplayComponent = AdapterFieldGroup
      object CmdFirstRow: TAdapterActionButton
        ActionName = 'FirstRow'
        Caption = '<<'
      end
      object CmdPrevRow: TAdapterActionButton
        ActionName = 'PrevRow'
        Caption = '<'
      end
      object CmdNextRow: TAdapterActionButton
        ActionName = 'NextRow'
        Caption = '>'
      end
      object CmdLastRow: TAdapterActionButton
        ActionName = 'LastRow'
        Caption = '>>'
      end
      object CmdNewRow: TAdapterActionButton
        ActionName = 'NewRow'
        Caption = 'Novo'
      end
      object CmdEditRow: TAdapterActionButton
        ActionName = 'EditRow'
        Caption = 'Editar'
      end
      object CmdDeleteRow: TAdapterActionButton
        ActionName = 'DeleteRow'
        Caption = 'Delete'
      end
      object CmdCancel: TAdapterActionButton
        ActionName = 'Cancel'
        Caption = 'Cancelar'
      End
    End
  End
End

```

```
    object CmdApply: TadapterActionButton
      ActionName = 'Apply'
      Caption = 'Aplicar'
    end
  end
end
end
end
end
```

## 4 EXTENSIVE MARKUP LANGUAGE (XML)

A *Extensive Markup Language* (XML) é uma linguagem de marcação de dados (*meta-markup language*) que provê um formato para descrever dados estruturados. Isso facilita declarações mais precisas do conteúdo e resultados mais significativos de busca através de múltiplas plataformas. A XML também permite o surgimento de uma nova geração de aplicações de manipulação e visualização de dados via internet (Laurent, 1999).

De acordo com Pitts-Moultis (2000), a XML é um subconjunto da *Standard Generalized Markup Language* (SGML), que é um padrão internacional para definir descrições de estruturas e conteúdo de diferentes tipos de documentos eletrônicos. Sua principal vantagem é permitir que um SGML genérico seja disponibilizado, recebido e processado através da internet.

A XML não é uma simples linguagem de marcação pré-definida, mas sim uma metalinguagem – uma linguagem que descreve outras linguagens – que permite definir sua própria marcação (Laurent, 1999).

De acordo com Marchal (2000), a XML é definida pelas seguintes especificações:

- a) *Extensive Markup Language* (XML) 1.0: define a sintaxe da XML;
- b) *XML Pointer Language* (XPointer) e *XML Linking Language* (XLink): define um padrão para representar os *links* entre os recursos. Além dos links simples, como a tag <A> da HTML, a XML possui mecanismos para ligar recursos múltiplos e diferentes. A XPointer descreve como endereçar um recurso, e a XLink descreve como associar dois ou mais recursos;
- c) *Extensive Style Language* (XSL): define a linguagem de folhas de estilos padrão para a XML.

### 4.1 PONTOS FORTES DA XML

De acordo com Laurent (1999), os pontos fortes da XML são:

- a) **inteligência:** a XML é inteligente para qualquer nível de complexidade. A marcação pode ser alterada de uma marcação mais geral como "<CÃO> Lassie </CÃO>" para uma mais detalhista, como "<CÃO> <VENHA\_PARA\_CASA> <COLLIE> Lassie </COLLIE> </VENHA\_PARA\_CASA> </CÃO>". A

informação conhece a si mesma. Não é necessária mais nenhuma idéia indesejável;

- b) adaptação: a XML é a linguagem mãe de outras linguagens. Marcações personalizadas podem ser criadas para qualquer necessidade. Se uma marcação que descreva como uma pizza *pepperoni* é diferente de uma pizza calabresa for necessária, ela pode ser feita;
- c) manutenção: a XML é fácil de manter. Ela contém somente idéias e marcações. Folhas de estilos e links vêm em separado, e não escondidas no documento. Cada um pode ser alterado separadamente quando for preciso com fácil acesso e fáceis mudanças;
- d) permite a definição de suas próprias *tags* conforme a necessidade do usuário;
- e) aplicações padronizadas para XML possibilitam que diferentes aplicativos trabalhem em conjunto trocando informações, proporcionando maior interoperabilidade;

## 4.2 OBJETIVOS

O comitê regulador da XML, denominado do *World Wide Web Consortium* (W3C) definiu quais os recursos que esta linguagem deve oferecer, definindo as seguintes diretrizes (Pitts-Moultis, 2000):

- a) deve ser utilizado de forma direta e objetiva na internet;
- b) suportar uma variedade de aplicativos;
- c) ser compatível com o SGML;
- d) deve ser fácil o bastante para escrever programas que processem documentos XML;
- e) possuir pouco ou nenhum recurso adicional;
- f) os documentos necessitam ser legíveis e relativamente claros;
- g) o projeto de XML deve ser capaz de ser preparado rapidamente;
- h) o projetista do XML deve ser formal e conciso;
- i) os documentos devem ser fáceis de serem criados;
- j) a concisão na marcação é de pouca importância.

## 4.3 DOCUMENTOS

Todos os documentos XML seguem as mesmas diretrizes estruturais. Isso facilita a criação a partir do momento em que entender as necessidades do aplicativo associado. O Quadro 9 apresenta um documento XML simples.

QUADRO 9 EXEMPLO DE XML

```
<?xml version="1.0" ?>
<Livro>
  <Titulo>XML</Titulo>
  <Autor>Fabiano</Autor>
  <Resumo/>
</Livro>
```

O documento começa com uma instrução de processamento: `<?xml ...?>`. Esta é a declaração XML. Embora não seja obrigatória, a sua presença explícita identifica o documento como um documento XML e indica a versão da XML com a qual ele foi escrito. Não há declaração do tipo do documento. Diferentemente da SGML, a XML não requer uma declaração de tipo de documento. Entretanto, uma declaração de tipo de documento pode ser fornecida.

Elementos vazios (`<Resumo/>` neste exemplo) têm uma sintaxe modificada. Enquanto que a maioria dos elementos em um documento envolvem algum conteúdo, elementos vazios são simplesmente marcadores onde alguma coisa ocorre. O final `</>` na sintaxe modificada indica a um programa que processa o documento XML que o elemento é vazio e uma marca de fim correspondente não deve ser procurada. Visto que os documentos XML não requerem uma declaração de tipo de documento, sem esta pista seria impossível para um analisador XML determinar quais marcas são intencionalmente vazias e quais teriam sido deixadas vazias por um erro.

Os documento XML são compostos de marcas e conteúdos. Existem seis tipos de marcações que podem ocorrer em um documento XML: elementos, referências a entidades, comentários, instruções de processamento, seções marcadas e declarações de tipos de

documento. As seções seguintes introduzem cada um destes conceitos de marcação (Pitts-Moutis, 2000).

### 4.3.1 ELEMENTOS

Elementos são a forma de marcação mais comum. São delimitados pelos sinais de menor e maior. A maioria dos elementos identificam a natureza do conteúdo que envolvem. Alguns elementos podem ser vazios, como visto anteriormente; neste caso eles não têm conteúdo. Se um elemento não é vazio, ele inicia com uma marca de início “<element>”, e termina com uma marca de término, “</element>” (Laurent, 1999).

### 4.3.2 ATRIBUTOS

Atributos são informações referentes à exibição ou apresentação da informação de um elemento. Os atributos ocorrem após o nome do elemento, como pode ser observado no Quadro 10.

QUADRO 10 EXEMPLO DE ATRIBUTO

<pre>&lt;Dicionario Nome="exemplo1.mdl"&gt;</pre>
---

Em XML, todos os valores de atributos devem estar entre aspas.

### 4.3.3 REFERÊNCIAS A ENTIDADES

A fim de introduzir a marcação em um documento, alguns documentos foram reservados para identificar o início da marcação. O sinal de menor (“<”), por exemplo, identifica o início de um elemento. Para inserir estes caracteres no documento como conteúdo, entidades são utilizadas para representar estes caracteres especiais. As entidades também são usadas para referenciar um texto frequentemente repetido ou alterado e incluí-lo no conteúdo de arquivos externos (Laurent, 1999).

Cada entidade deve ter um nome único. Para usar uma entidade, deve simplesmente referenciá-la pelo nome. As referências às entidades iniciam com “&” e terminam com um “;”.



Por exemplo, a entidade “lt” insere um literal “<” em um documento. A cadeia de caracteres “<element>” pode ser representada em um documento XML como “&lt;element>”.

Uma forma especial de referência a entidades, chamada de referência a caractere, pode ser usada para inserir arbitrariamente caracteres *unicode* no documento. Este é um mecanismo para inserir caracteres que não podem ser diretamente digitados pelo seu teclado.

#### 4.3.4 COMENTÁRIOS

Comentários iniciam com o literal “<!--” e terminam com “-->”. Os comentários podem conter qualquer dado, exceto a literal "--”. Comentários não fazem parte de um conteúdo textual de um documento XML. Um processador XML não é preciso para reconhecê-los na aplicação.

#### 4.3.5 INSTRUÇÕES DE PROCESSAMENTO

Instruções de processamento (PIs) são formas de fornecer informações a uma aplicação. Assim como os comentários, elas não são textualmente parte de um documento XML, mas o processador XML pode reconhecê-las na aplicação.

As instruções de processamento têm a forma: “<?nome dadospi?>”. O nome, chamado de alvo PI, identifica a PI na aplicação. As aplicações processam somente os alvos que eles reconhecem e ignoram todas as outras PIs. Qualquer dado que segue o alvo PI é opcional (Marchal, 2000).

#### 4.3.6 SEÇÕES CDATA

Em um documento, uma seção CDATA instrui o analisador para ignorar os caracteres de marcação. Considerar um código-fonte em um documento XML. Ele pode conter caracteres que o analisador XML iria normalmente reconhecer como marcação (“<” e “&”, por exemplo). Para prevenir isto, uma seção CDATA pode ser usada.

Entre o início da seção, “<![CDATA[“, e o fim da seção, “]]>”, todos os dados de caracteres são passados diretamente para a aplicação, sem interpretação. Elementos, referências a entidades, comentários e instruções de processamento são todos irreconhecíveis

e os caracteres que os compõem são passados literalmente para a aplicação (Laurent, 1999). A única cadeia de caracteres que não pode ocorrer em uma seção CDATA é "]]>".

### **4.3.7 DEFINIÇÃO DE TIPO DE DOCUMENTO**

Um *Document Type Definition* (DTD), fornece um conjunto de regras para o documento XML. As DTDs definem as instruções para a estrutura do documento XML e definem quais elementos serão usados ao longo do documento. Mais especificamente uma DTD especifica (Pitt-Moutis, 2000):

- a) os tipos de elementos que serão permitidos dentro do documento XML;
- b) as diversas características de cada tipo de elemento juntamente com os atributos usados;
- c) quaisquer notações que possam ser encontradas dentro de um documento;
- d) as entidades que podem ser usadas dentro de um documento.

## 5 FERRAMENTAS

Neste capítulo serão apresentadas as ferramentas e técnicas utilizadas na construção do protótipo da ferramenta.

### 5.1 DELPHI 6.0

O Borland Delphi 6 é um ambiente de programação para o sistema operacional Windows que suporta o padrão CORBA.

Os principais recursos do produto incluem *BizSnap*, *WebSnap* e *DataSnap*, que permitem aos desenvolvedores criar aplicações que simplifiquem a integração *Bussines to Bussines* (B2B) com suporte a XML, *Simple Object Access Protocol* (SOAP) e WSDL.

O suporte a *WebServices* possibilita integração com novos *WebServices* baseados em plataformas como *.Net* e *BizTalk* da Microsoft e *ONE* da Sun Microsystems. Além disso, quando associado ao Kylix, os usuários do Delphi 6 podem criar aplicações de fonte única tanto para Windows quanto para Linux (Inprise, 2001).

Para suporte ao padrão CORBA o Delphi 6.0 provê os seguintes mecanismos:

- a) um compilador IDL para pascal (idl2pas.jar);
- b) duas *units* para auxiliar o desenvolvimento dos objetos (OrbPas40.pas e Corba.pas);
- c) *runtime library* (orbpas40.dll, orb\_br.dll, vport\_dll).

### 5.2 KYLIX 2

O Kylix2 é um ambiente de programação para o sistema operacional Linux, que utiliza para o desenvolvimento das aplicações um conjunto de bibliotecas denominado *Cross Platform* (CLX). Estas bibliotecas são portáveis para os sistemas operacionais Windows e Linux (Inprise, 2001). Isso permite que uma aplicação escrita com CLX seja compilada no Delphi 6 ou no Kylix 2.

Para acesso a banco de dados foi criado uma nova *engine* denominada de dbExpress, também implementado no Delphi 6. O dbExpress é uma camada muito fina para acesso

unidirecional ao banco de dados. Este fato torna o acesso mais rápido em comparação ao acesso utilizando *drivers* ODBC (Inprise, 2001).

A utilização do Kylix 2 para este trabalho, deve-se ao fato de possuir suporte ao padrão CORBA, o que não acontecia na sua primeira versão.

Para suporte ao padrão CORBA o Kylix 2 provê os seguintes mecanismos:

- d) um compilador IDL para pascal (idl2pas.jar);
- e) duas *units* compiladas para auxiliar o desenvolvimento dos objetos (OrbPas40.dcu e Corba.dcu);
- f) *runtime library* (liborbpas45.so.1, liborbpas45.so, liborbpas45.so.0.0.1).

### 5.3 VISIBROKER

Visibroker é uma implementação do padrão CORBA 2.3 desenvolvido pela *Inprise Corporation*. Para este trabalho foi utilizado a versão 4.0 que utiliza apenas chamadas estáticas para comunicação entre os objetos CORBA.

Esta implementação do CORBA possui os seguintes recursos (Inprise, 2001):

- a) geração de objetos servidores;
- b) suporte todos os tipo básicos CORBA (*int, float, double, string, etc*), exceto o tipo *fixed*, porque não existe tipo equivalente em pascal;
- c) suporte a módulos;
- d) heranças de interface simples ou múltiplas;
- e) tipo compostos de dados: *Structs, Unions, Arrays e Sequences*;
- f) passagem de parâmetro por *in* (entrada), *out* (saída) ou *inout* (entrada/saída);
- g) operações *one-way* e *two-way*;
- h) atributos;
- i) exceções;
- j) constantes;
- k) módulos embutidos;
- l) tipo anônimo;
- m) tipo CORBA *Object*;
- n) tipo CORBA *Any*;

- o) *Callbacks*;
- p) serviço de nomes;
- q) serviço de eventos.

### 5.3.1 EXEMPLO DE UM PROGRAMA DISTRIBUÍDO

O exemplo a seguir mostra como é desenvolvido uma aplicação distribuída. Para o exemplo foi utilizada a versão 4 do Visibroker com o ambiente de desenvolvimento Delphi 6.0. O primeiro passo para criação um programa distribuído é definir sua IDL. Como exemplo será utilizado a IDL do Quadro 11.

QUADRO 11 IDL DE EXEMPLO

```
interface Calculadora
{
    double getSoma(in double num1, in double num2);
};
```

Após a definição da IDL é necessário salvá-la, neste caso foi utilizado o nome “calc.idl”. O próximo passo é compilar a IDL para *Object Pascal* para gerar os clientes *Stub* e os servidores *Skeletons* a partir do arquivo IDL. O Quadro 12, apresenta o comando utilizado nesta operação.

QUADRO 12 COMPILADOR IDL

```
Idl2pas [arguments] file
```

Fonte: Inprise (2001)

Após a execução deste comando são gerados quatro arquivos (*units*):

- a) calc\_c.pas: servidor *skeleton*;
- b) calc\_s.pas: cliente *stub*;
- c) calc\_i.pas: contém a definição da interface de calculadora;
- d) calc\_impl.pas: este arquivo é utilizado para implementação do objeto servidor.

O próximo passo é implementar o objeto distribuído. Esta implementação deve ser feita na *unit calc\_impl.pas*. Um exemplo é mostrado no Quadro 13.

### QUADRO 13 EXEMPLO DE UMA IMPLEMENTAÇÃO DE UM OBJETO

```

Unit calc_impl;

interface

uses
  SysUtils,
  CORBA,
  calc_i,
  calc_c;

type
  TCalculadora = class;

  TCalculadora = class(TInterfacedObject, calc_i.Calculadora)
  Protected
  Public
    constructor Create;
    function getSoma ( const num1 : Double;
                       const num2 : Double): Double;

  end;

implementation

constructor TCalculadora.Create;
begin
  inherited;
end;

function TCalculadora.getSoma(const num1:Double;
                              const num2: Double): Double;
begin
  result := num1 + num2;
end;

initialization
end.

```

A seguir deve-se criar a *unit* principal, onde é necessário inicializar o CORBA e associar as *units* da interface, implementação e *skeleton* do objeto distribuído (Quadro 14).

### QUADRO 14 PROGRAMA PRINCIPAL

```

unit uServer;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Forms,
  {units necessárias para a aplicação distribuída}

```

```

Corba,
calc_I, calc_S, calc_Impl;

type
TFrmCalcServer = class(TForm)
  procedure FormCreate(Sender: TObject);
protected
  fCalc : Calc; // declaração do objeto
  procedure InitCorba; // faz a inicialização do ORB
public
  { public declarations }
end;

var
  FrmCalcServer: TFrmCalcServer;

implementation

{$R *.DFM}

procedure TFrmCalcServer.InitCorba;
begin
  CorbaInitialize; //inicializa o orb

  // Cria o objeto servidor
  fCalc := TCalckeleton.Create('Calc Server', TCalc.Create);
  BOA.ObjIsReady(Calc as _Object); // disponibiliza o objeto
                                   // utilizando o Basic
                                   // Object Adapter
end;

procedure TFrmCalcServer.FormCreate(Sender: TObject);
begin
  InitCorba;
end;

end.

```

Depois de criado o servidor é necessário criar o cliente. Para isso deve-se inicializar o CORBA, associar as *units* da *interface* do objeto servidor e a *unit stub* e efetuar a ligação com o objeto servidor (*bind*), como pode ser observado no Quadro 15.

#### QUADRO 15 APLICAÇÃO CLIENTE

```

unit uClient;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  {units Corba}
  Corba, calc_I, calc_C;

```

```

Type
  TFrmCliente = class(TForm)
    edV1: TEdit;
    edV2: TEdit;
    btResult: TButton;
    edResult: TEdit;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
  protected
    fCalc : Calc;
    procedure InitCorba;
  public
  end;

var
  FrmCliente: TfrmCliente;

Implementation

{$R *.DFM}

procedure TFrmCliente.InitCorba;
var
  a: real;
begin
  //inicializa CORBA
  CorbaInitialize;
  fCalc := TCalcHelper.bind;
end;

procedure TfrmCliente.FormCreate(Sender: TObject);
begin
  InitCorba;
End;

procedure TfrmCliente.btResultClick(Sender: TObject);
begin
  // chama o método do servidor getSoma
  edResult.Text:= FloatToStr( fCalc.getSoma( StrToFloat(edV1.Text),
                                             StrToFloat(edV2.Text) ));
end;

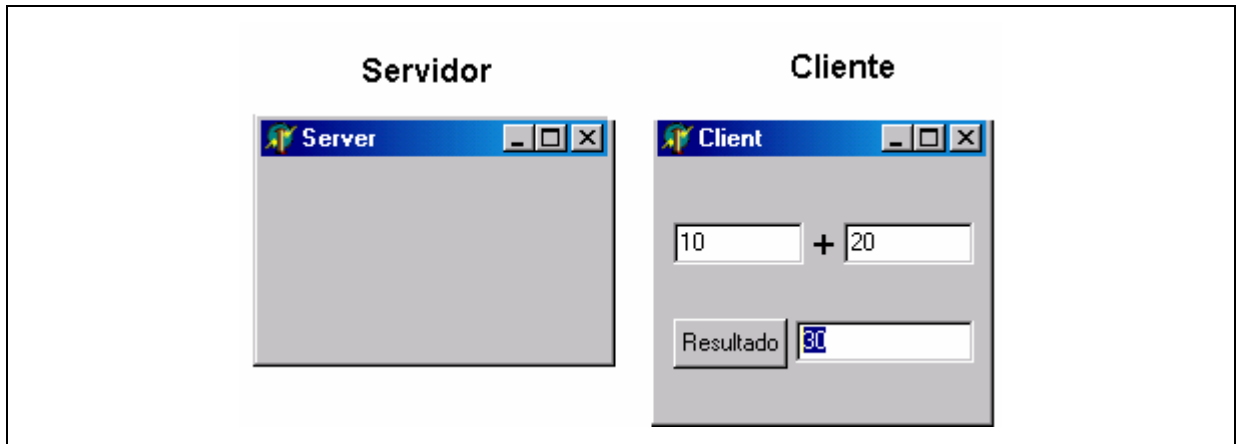
End.

```

Depois de compilar a aplicação, é necessário inicializar o CORBA, executando o comando “OsAgent” do Visibroker em cada uma das máquinas onde será executados os programas CORBA. Para isso é necessário apenas executar o comando: “start osagent”. A aplicação executando pode ser observada na Figura 10.



FIGURA 10 APLICAÇÃO EXEMPLO



Como o Visibroker não é um produto desenvolvido em *object pascal*, de acordo com Inprise(2001), uma aplicação possui um desempenho cerca de 10% menor comparando com uma aplicação escrita em C++.

## 6 DESENVOLVIMENTO

Neste capítulo são apresentadas as especificações do protótipo, juntamente com a implementação do mesmo.

### 6.1 REQUISITOS PRINCIPAIS DO PROBLEMA

Os requisitos identificados para este trabalho foram:

- a) ser capaz de gerar um dicionário de dados com base nos arquivos gerados pela ferramenta Rational Rose 2000, que contenha uma especificação UML e dele extrair os diagramas de classes. A utilização do Rational Rose deve-se ao fato de ser uma ferramenta muito utilizada para modelagem utilizando a linguagem UML;
- b) poder salvar a especificação em um arquivo XML para que esta possa ser utilizada por outros sistemas, uma vez que a linguagem XML é um padrão e possui um manuseio mais fácil que uma especificação do Rational Rose;
- c) permitir aplicar alguns *Design Patterns* disponíveis, sobre o dicionário;
- d) poder gerar o *script* para um banco de dados relacional com base no diagrama de classes;
- e) gerar objetos (distribuídos) de negócios e sua ligação com os objetos de acesso a dados;
- f) gerar um cliente simples, para interação do usuário com os objetos distribuídos;

### 6.2 ESPECIFICAÇÃO DO PROTÓTIPO

A especificação do sistema foi feita utilizando-se alguns diagramas da UML - *Unified Modeling Language*. A ferramenta CASE utilizada para esta especificação foi o *Rational Rose 2000* da Rational®. A seguir serão descritos alguns diagramas.

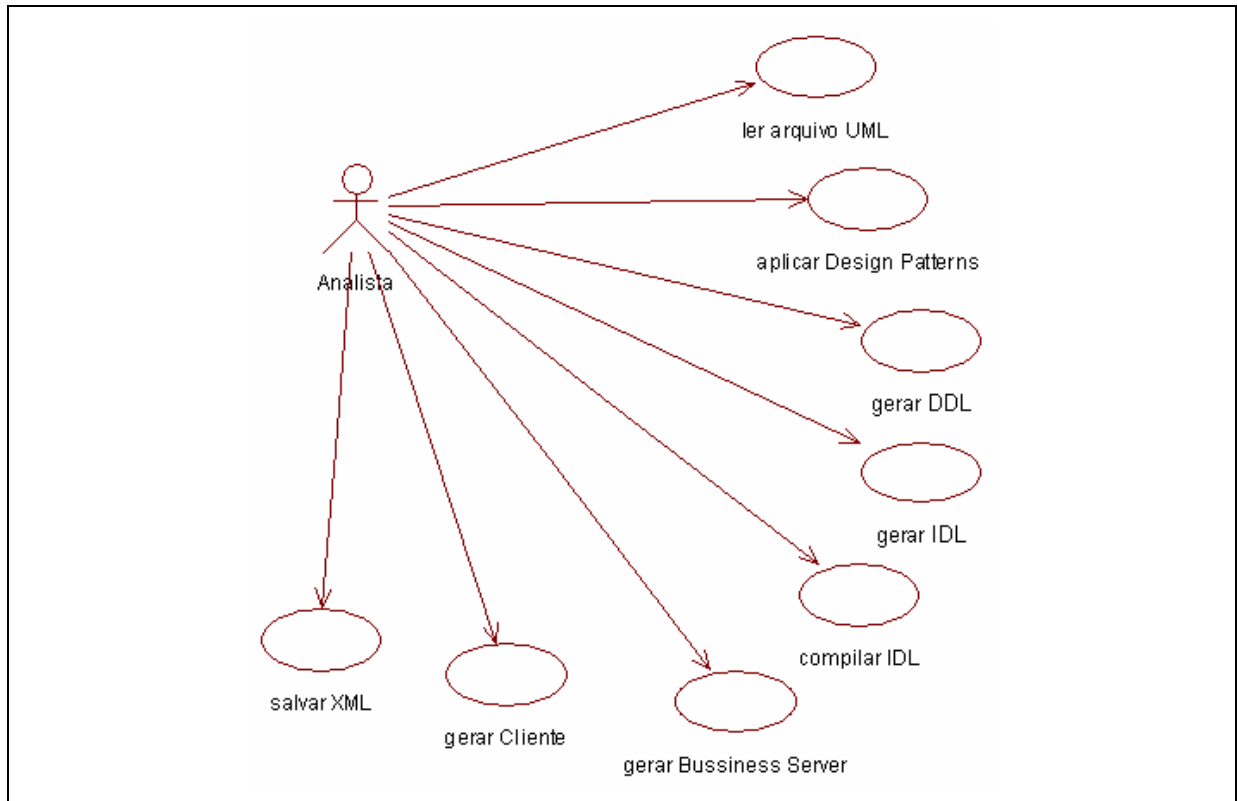
#### 6.2.1 DIAGRAMA DE CASOS DE USO

Neste protótipo foram observados os seguintes diagramas de casos de uso primários (Figura 11):

- a) ler arquivo UML: o analista seleciona um arquivo do Rational Rose 2000 e o protótipo gera uma estrutura de classes correspondente;

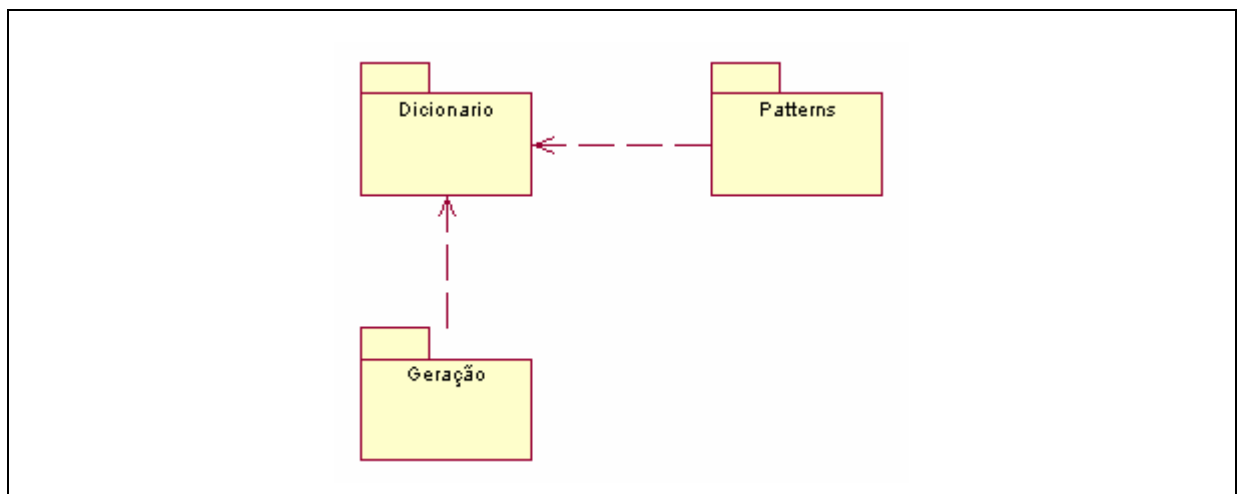
- b) aplicar *Design Patterns*: sobre o dicionário de dados é possível aplicar alguns padrões com base na estrutura de classes gerada pelo caso de uso “ler arquivo UML”;
- c) gerar *data definition language* (DDL): gera o *script* e faz a criação do banco de dados do sistema;
- d) gerar IDL: gera as interfaces para que os objetos distribuídos possam se comunicar;
- e) compilar IDL: faz o mapeamento de uma IDL para a linguagem *object pascal* para definir as interfaces dos métodos que serão implementados;
- f) gera *business server*: gera a implementação do servidor de objetos de negócio que pode conter um ou mais objetos distribuídos e também faz a ligação com a camada de acesso aos dados;
- g) gerar cliente: com o mapeamento gerado pelo caso de uso “compilar IDL” mais as informações adicionadas no caso de uso “aplicar *Design Patterns*”, o protótipo gera uma aplicação cliente para comunicar-se com os objetos da camada de negócio;
- h) salvar XML: salva as definições do dicionário em um arquivo XML.

FIGURA 11 DIAGRAMA DE CASOS DE USO



## 6.2.2 DIAGRAMA DE CLASSES

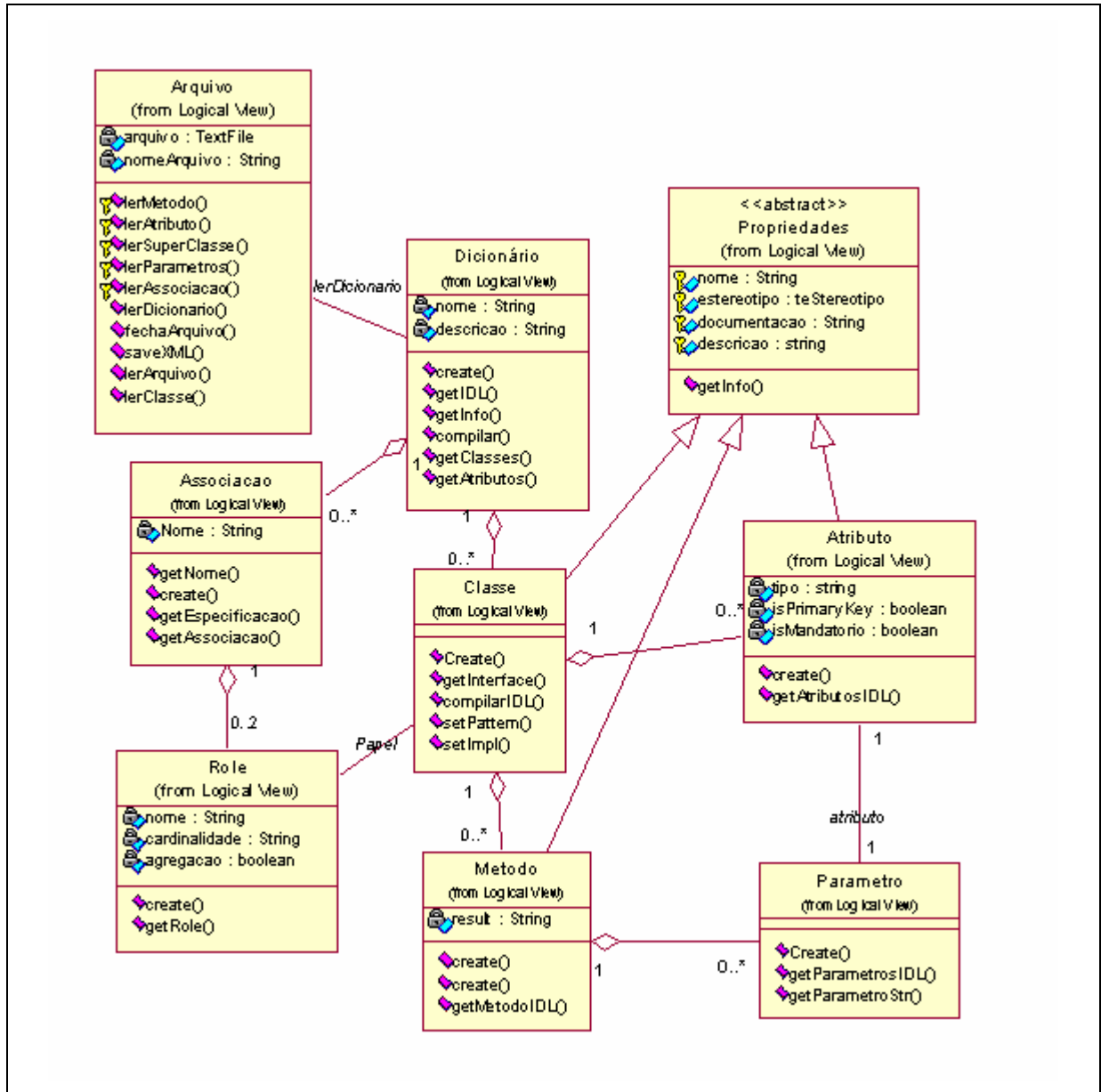
As classes identificadas para este sistema estão separadas em *packages*, sendo elas: dicionário, *patterns* e geração (Figura 12). A seguir são descritos cada uma desta *packages*.

FIGURA 12 *PACKAGES*

### 6.2.2.1 PACKAGE DICIONÁRIO

As classes que fazem parte da *package* dicionário são apresentadas na Figura 13.

FIGURA 13 DIAGRAMA DE CLASSES DA *PACKAGE* DICIONÁRIO



As classes possuem as seguintes finalidades:

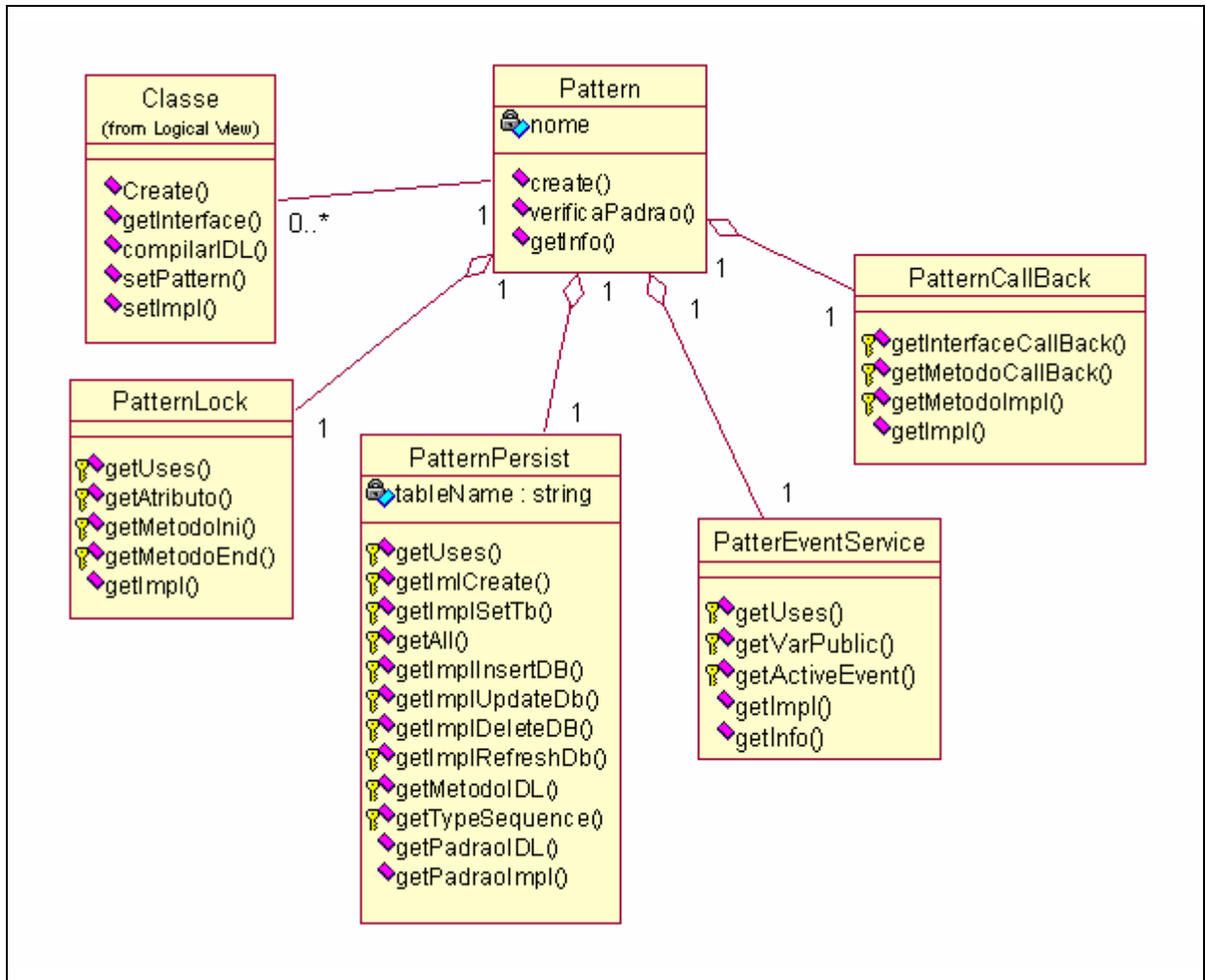
- arquivo: é responsável pela manipulação de arquivos, tanto a leitura do arquivo UML como salvar a especificação em XML;
- propriedades: é uma classe abstrata que contém informações comuns para classes,

- métodos e atributos;
- c) dicionário: é responsável pelo gerenciamento do dicionário. Com isso é possível manipular diversas especificações UML. Na realidade cada arquivo que é aberto corresponde a um dicionário;
  - d) classe: manipula as classes extraídas do arquivo UML;
  - e) método: é responsável pelos métodos extraídos do arquivo UML
  - f) parâmetros: representa os parâmetros dos métodos lidos;
  - g) atributo: é responsável pelos atributos das classes e também pelos parâmetros dos métodos;
  - h) associação: são responsáveis pelas associações entre as classes do diagrama de classes;
  - i) *role*: para cada associação, existem duas roles, cada uma delas contém a informação da classe a qual representa, bem como informações como cardinalidade e se a associação é uma agregação ou não.

### **6.2.2.2 PACKAGE PATTERNS**

Esta *package* tem a finalidade de aplicar os conceito de *Design Patterns* apresentados anteriormente sobre uma implementação em *object pascal* e/ou definição de *interfaces* IDL. As classes que fazem parte desta *package* são apresentadas na Figura 14

FIGURA 14 PACKAGE PATTERNS



As classes possuem as seguintes finalidades:

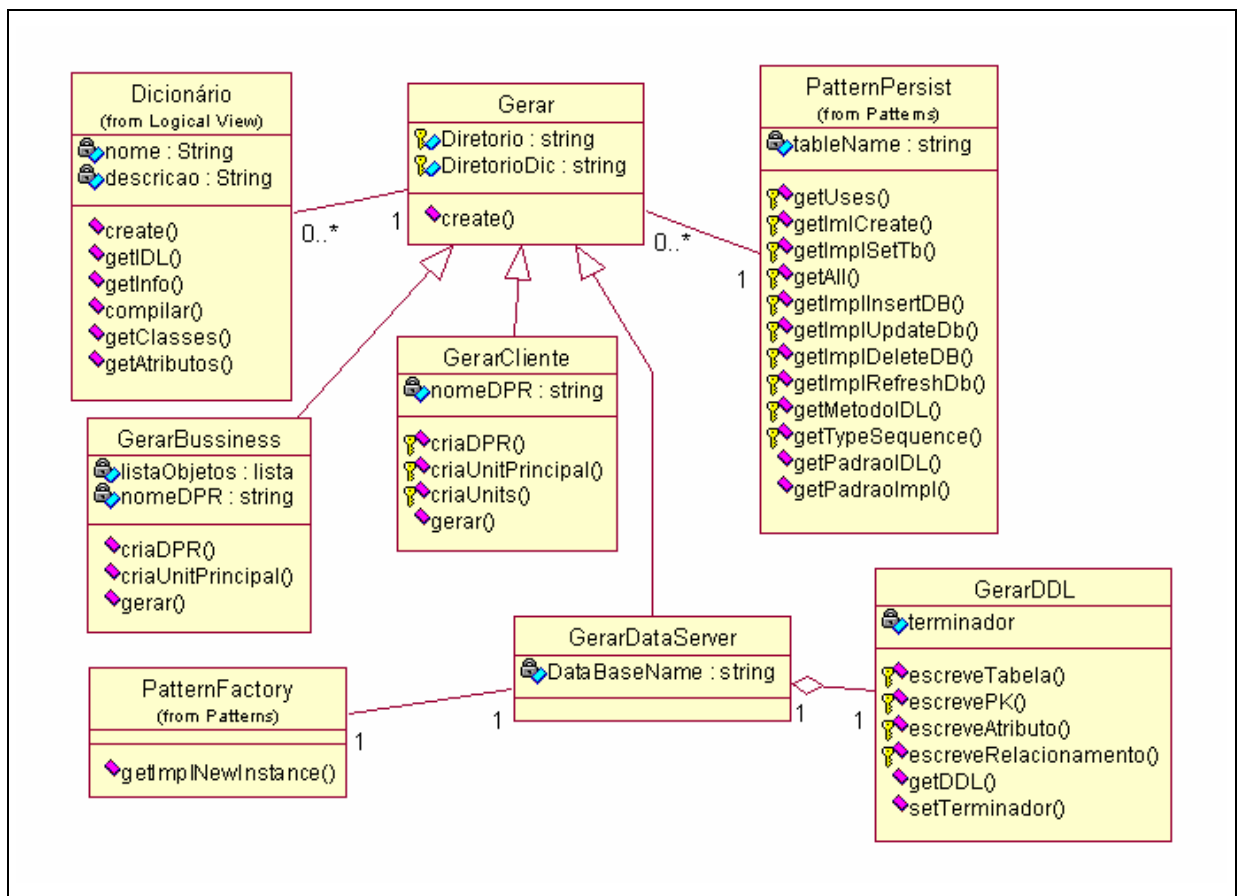
- pattern*: esta classe é responsável por verificar quais padrões serão utilizados e após isso delega a aplicação do padrão às outras classes de *patterns*;
- patternLock*: esta classe é responsável pela implementação de um *Design Pattern Lock*, sendo que este padrão refere-se apenas a implementação em *object pascal*;
- patternPersist*: é responsável pela utilização do *Design Pattern Persistent Layer*, sendo que este padrão refere-se tanto a definição do IDL como da implementação;
- patternEventService*: tem a função de incorporar a implementação de um *Design Pattern* notificação;
- patternCallBack*: esta classe aplica o *Design Pattern CallBack*, que incorpora funcionalidades tanto da *interface IDL* como para implementação.

Os padrões *factory* e *template* não aparecem no diagrama de classes, pois não são padrões que serão utilizados nos objetos de negócio, sendo que o padrão *factory* somente será utilizado para prover acesso a um banco de dados relacional e o *template* apenas na geração da interface WEB.

### 6.2.2.3 PACKAGE GERAÇÃO

Esta *package* tem a finalidade de gerar os programas principais dos objetos de negócio bem como as conexões com os objetos de acesso a dados e implementam também um cliente para acesso às informações. Seu diagrama de classes é apresentado na Figura 15.

FIGURA 15 PACKAGE GERAÇÃO



As classes possuem as seguintes finalidades:

- gerar: esta classe possui algumas informações necessárias para a geração do código fonte, como diretório da aplicação e diretório das bibliotecas;



- b) gerarBusiness: esta classe é responsável por gerar o programa principal dos objetos de negócio, sendo que é possível escolher quais objetos serão parte deste servidor bem como replicar um objeto em diversos servidores;
- c) gerarCliente: esta classe é responsável por criar uma aplicação WEB, para interação com o usuário;
- d) gerarDDL: esta classe tem a responsabilidade de gerar a definição da base dados;
- e) gerarDataServer: esta classe tem a responsabilidade de gerar o banco de dados com base na DDL gerada pela classe “gerarDDL”.

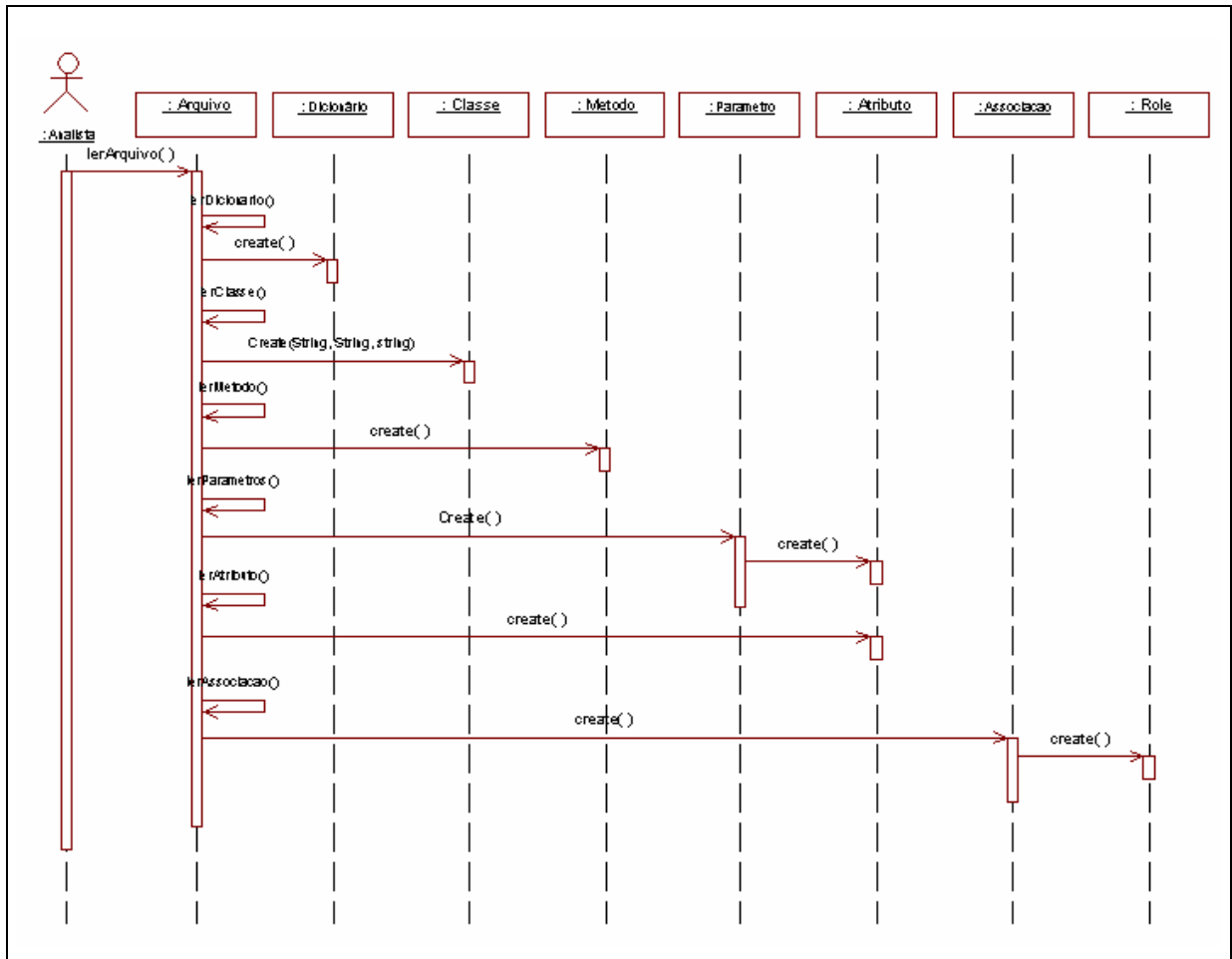
### **6.2.3 DIAGRAMA DE SEQÜÊNCIA**

Os diagramas de seqüências representam, como o próprio nome indica, a seqüência em que as ações ocorrem dentro do sistema. Eles demonstram como é feita a troca de mensagens entre as classes (objetos). Então cada caso de uso definido na Figura 11 irá gerar um diagrama de seqüência correspondente.

#### **6.2.3.1 LER ARQUIVO UML**

Este diagrama inicia com a solicitação de leitura de um arquivo gerado pela ferramenta Rational Rose 2000, feita pelo analista, sendo posteriormente interpretado. A partir daí começa o processo de criação dos objetos em memória. Para cada arquivo lido é criado um objeto “dicionário” e adicionadas todas as classes, métodos, atributos e associações referentes àquele arquivo (Figura 16).

FIGURA 16 DIAGRAMA DE SEQÜÊNCIA LER ARQUIVO UML



Para a leitura de um arquivo que contém uma especificação na linguagem UML é necessário ter uma especificação deste arquivo. O Quadro 16 apresenta uma parte de um arquivo do Rational Rose que corresponde a especificação do diagrama de classes.

QUADRO 16 ARQUIVO DO RATIONAL ROSE 2000

```

(object Class "Pessoa"
  quid      "3BF48D120240"
  stereotype "persistent"
  operations (list Operations
    (object Operation "getInfo"
      quid      "3BFC57260281"
      parameters (list Parameters
        (object Parameter "codigo"
          type      "long"))
      result    "string"
      concurrency "Sequential"
      opExportControl "Public"
      uid      0))
  class_attributes (list class_attribute_list

```

```

(object ClassAttribute "codigo"
  quid      "3BF48D3402CB"
  type      "long")
(object ClassAttribute "nome"
  quid      "3BF48D3E00B3"
  type      "string")
(object ClassAttribute "fone"
  quid      "3BF48D460303"
  type      "string")
(object ClassAttribute "email"
  quid      "3BF48D4B02F7"
  type      "string"))
language   "Java")
(object Class "Cidade"
  quid      "3BFC568D038F"
  stereotype "persistent"
  class_attributes (list class_attribute_list
    (object ClassAttribute "codigo"
      quid      "3BFC569D0234"
      type      "long")
    (object ClassAttribute "nome"
      quid      "3BFC56B001CD"
      type      "string")))
  language   "Java")
(object Association "reside"
  quid      "3BFC56E003DF"
  roles     (list role_list
    (object Role "reside_pessoa"
      quid      "3BFC56E1021E"
      supplier   "Logical View::Pessoa"
      quidu      "3BF48D120240"
      client_cardinality (value cardinality "0..n")
      is_navigable      TRUE)
    (object Role "reside_cidade"
      quid      "3BFC56E1026E"
      supplier   "Logical View::Cidade"
      quidu      "3BFC568D038F"
      client_cardinality (value cardinality "1")
      is_navigable      TRUE)))

```

No arquivo descrito anteriormente existe uma classe chamada “Pessoa” que possui o estereótipo “*persistent*” com quatro atributos (código, nome, fone e email) e um método “*getInfo*” e uma classe chamada “Cidade” também com estereótipo “*persistent*” que possui os atributos (código e nome). Contém também uma associação chamada “reside” que interliga as duas classes.

O Quadro 17 apresenta a estrutura básica de um arquivo do Rational Rose.

## QUADRO 17 ESPECIFICAÇÃO DO LEITOR DE ARQUIVOS DO RATIONAL ROSE

```

(Class) -> (object Class(string) // nome da classe
  quid (ident) // identificador
  stereotype (string) // estereotipo
  operations (List) // lista da operações
  class_attributes (List) // atributos da classe
  superclasses(List) // herança
  language(string) // linguagem de programação
  exportControl(string) // visibilidade

(Operation) -> (objet Operation (string) // nome do método
  quid (ident) // identificador
  paramaters (List) // lista de parâmetros
  // da operação
  result (string) // retorno
  concurrency (string) // seqüencial
  opExportControl (string) // visibilidade
  stereotype (string) // estereotipo

(ClassAttribute) -> (objet ClassAttibute (string) // nome do
  // atributo
  quid (ident) // identificador
  type (string) // tipo do atributo
  exportControl (string) // visibilidade
  stereotpe (string) // estereotipo

(Associations) -> (object Association (string) // nome da
  // associação
  quid (ident) // identificador
  roles (Lista) // lista de papeis

(Role) -> (object Role (string) // nome do papel
  quid (ident) // identificador
  supplier (string) // associação com a classe
  quidu (ident) // id da associação
  is_navigable (boolean) // a associação é em ambas direções?
  is_aggragate (boolean) // é agregação
  client_cardinality (value) //cardinalidade

```

Onde as finalidades dos identificadores são observados a seguir:

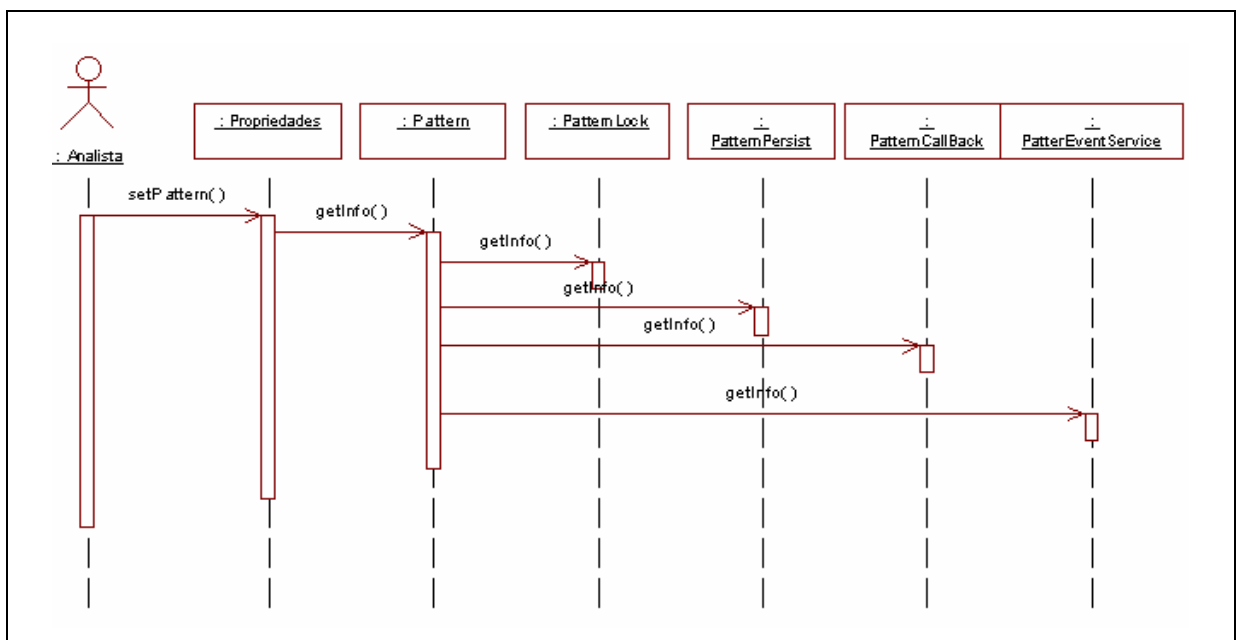
- a) *class*: este identificador contém a especificação de uma classe;
- b) *operation*: identifica os métodos da classe;
- c) *class\_attributes*: contém informações relacionadas aos atributos da classe;
- d) *association*: este identificador contém informações relacionadas as associações;
- e) *roles*: identifica informações sobre as *roles* (papéis ou atribuições que uma classe assume em uma associação);
- f) *role*: contém o nome da *role*;

Neste trabalho somente será utilizado o diagrama de classes que foi modelado na ferramenta CASE Rational Rose. As outras informações contidas no arquivo, como por exemplo, diagrama de casos de uso, diagrama de seqüência, entre outros, serão ignorados.

### 6.2.3.2 APLICAR DESIGN PATTERNS

Este processo descrito na Figura 17, tem início quando um ator analista envia uma mensagem para um objeto “propriedades”, informando qual o *Design Pattern* que deseja utilizar. Este padrão é validado, ou seja, é verificado se o padrão informado existe. Após isso a classe *pattern* requisita informações referentes ao padrão específico para ser visualizada pelo analista. Destaca-se que o método “setPatern” é um método polimorfo e possui comportamento distinto dependo tipo do objeto para que é enviado a mensagem.

FIGURA 17 DIAGRAMA DE SEQÜÊNCIA APLICAR *DESIGN PATTERNS*

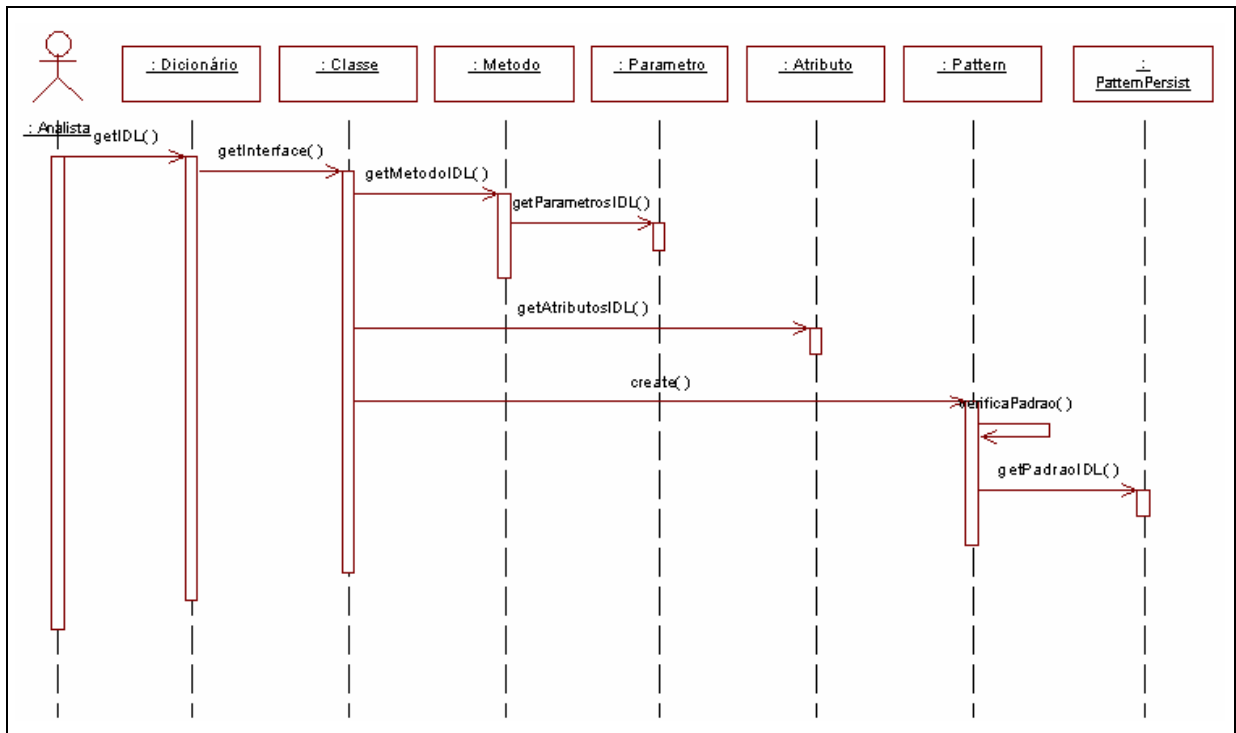


### 6.2.3.3 GERAR IDL

A Figura 18, mostra o diagrama de seqüência “gerarIDL”, que tem início com o envio da mensagem “getIDL” a um objeto “dicionário”. Então este objeto envia uma mensagem a todas as suas “classes” solicitando sua interface (getInteface). Cada objeto de classe solicita a seus métodos (getMetodoIDL) e atributos (getAtributoIDL) sua especificação IDL.

Finalmente é instanciado um objeto de “*pattern*”, onde ele verifica se foi aplicado algum padrão e se for o caso, ele adiciona a IDL a especificação do padrão. No protótipo os dois padrões que alteram à estrutura de uma IDL são o *persistent layer* e o *distributed callback*, uma vez que estes padrões influenciam a interface dos objetos.

FIGURA 18 DIAGRAMA DE SEQUÊNCIA GERAR IDL



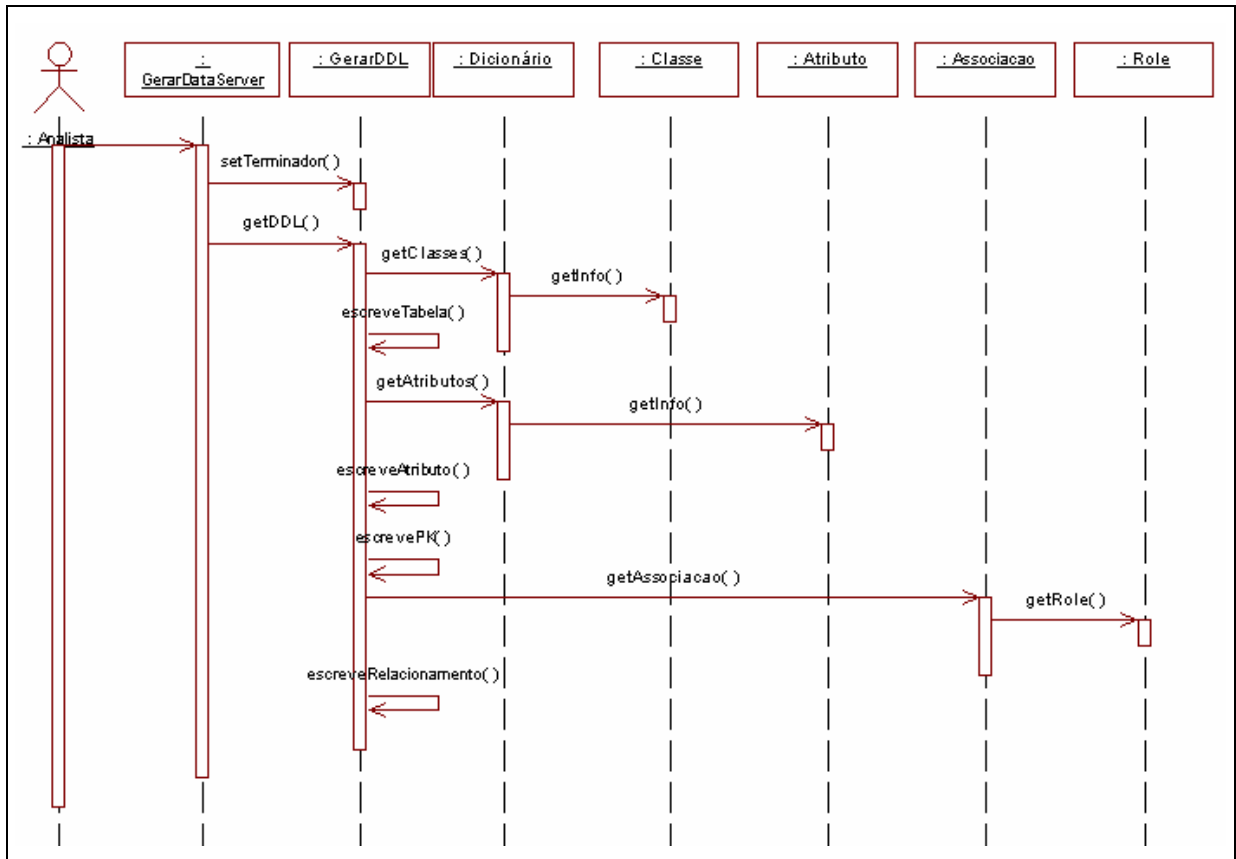
#### 6.2.3.4 GERAR DDL

Outro diagrama é o “Gerar DDL” identificado na Figura 19. Ele tem início com o envio da mensagens de configuração do terminador para o banco de dados. Após isso é enviado um método `getDDL`, passando como parâmetro o nome do banco de dados e o dicionário a gerar. Então inicia-se o processo de leitura dos objetos criados pelo processo “ler arquivo UML”, para criação do banco de dados relacional, que será utilizado no sistema resultante do protótipo.

O “`gerarDDL`” requisita ao objeto dicionário todas as suas classes e em seguida o processo de geração do arquivo efetivamente se inicia. Então são verificados todos os

atributos da classe para criação dos campos. Ao final são verificadas todas as associações para a criação das chaves estrangeiras e integridades referenciais.

FIGURA 19 DIAGRAMA DE SEQÜÊNCIA GERAR DDL



A implementação do “gerarDDL” é baseado no trabalho de Cunha (2001).

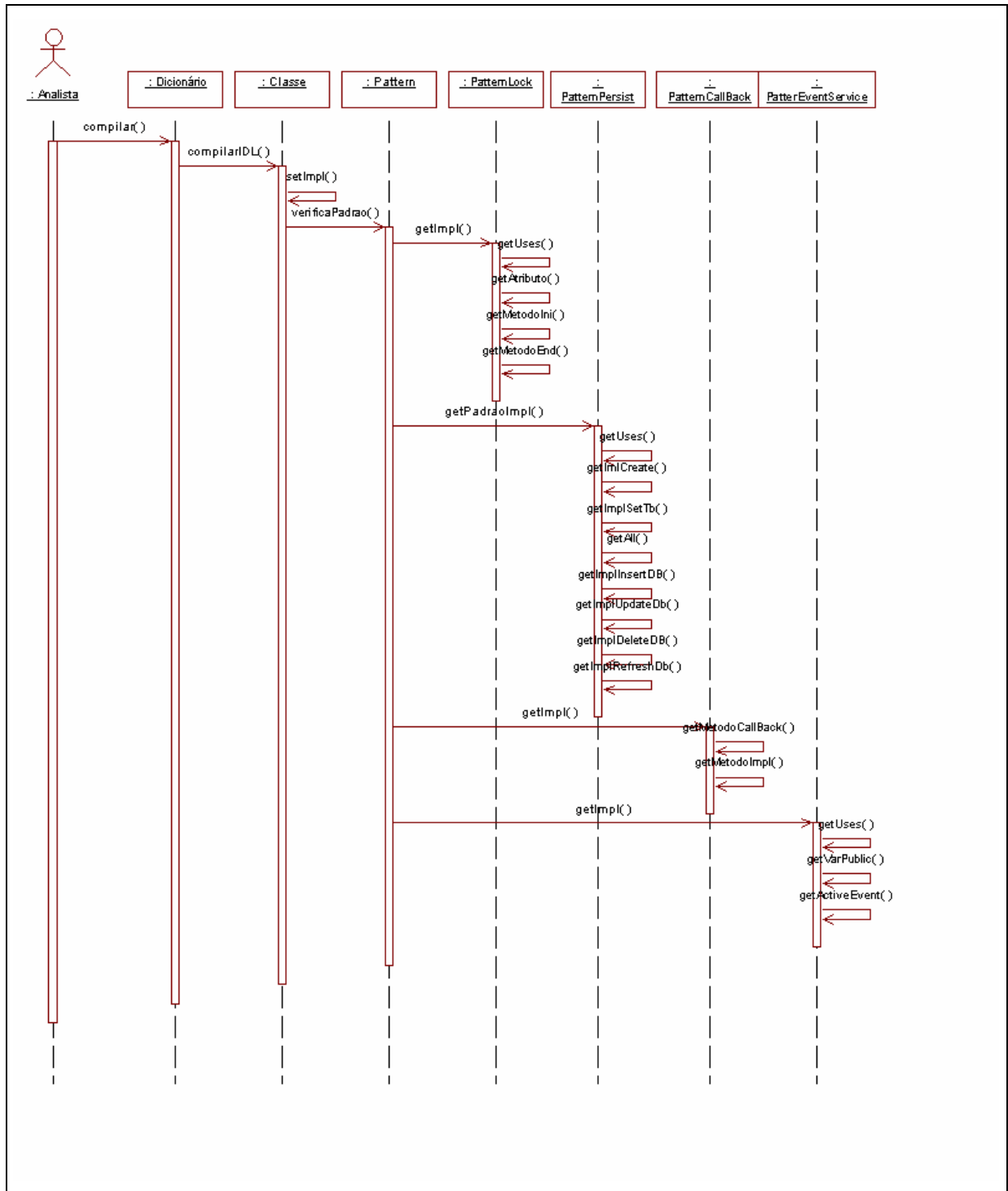
### 6.2.3.5 COMPILAR IDL

O processo compilar IDL identificado na Figura 20, inicia com o envio da mensagem “compilar” a um objeto “dicionário”. Então para cada classe do dicionário é enviado uma mensagem “compilarIDL”. Este método chama o compilador *idl2pas* que faz o mapeamento da IDL para *Object Pascal*.

Depois de compilada a IDL e gerado o mapeamento em *Object Pascal* é verificado se existe algum padrão associado através da mensagem “verifica padrão” a um objeto *pattern*. Se

possuir algum padrão, o objeto *pattern* delega a responsabilidade de adicionar a implementação do padrão para a implementação do objeto distribuído.

FIGURA 20 DIAGRAMA DE SEQÜÊNCIA COMPILAR IDL

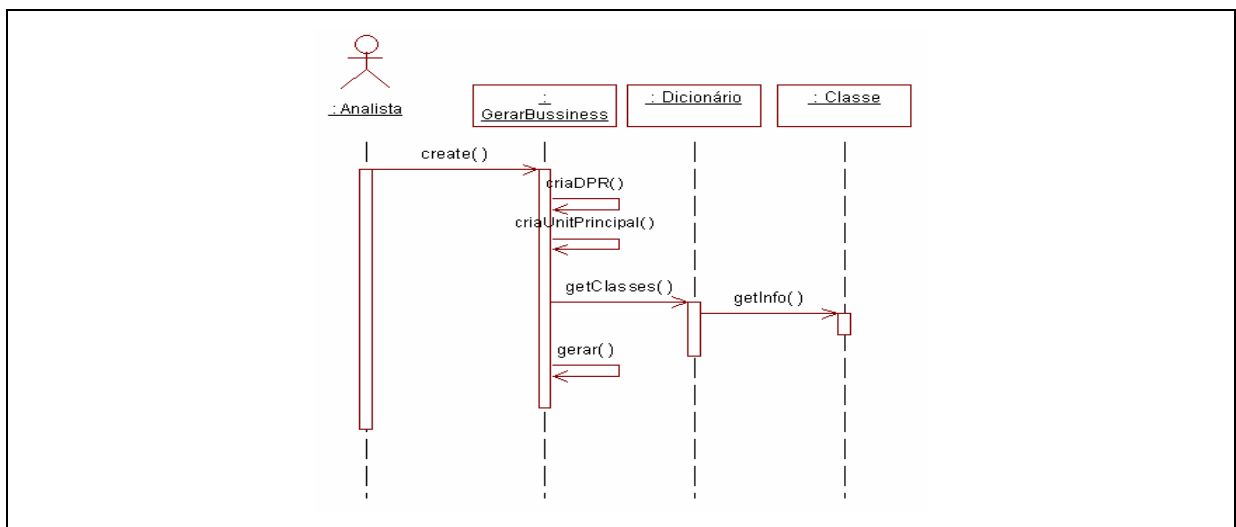




### 6.2.3.6 GERAR BUSINESS SERVER

O processo gerar *business server*, identificado na Figura 21, inicia com a criação de um objeto “*gerarBusiness*”, que é responsável pela criação do programa principal. Depois é necessário agregar à aplicação os objetos distribuídos gerados. Para isso ele envia uma mensagem para um objeto “dicionário” requisitando suas classes que foram selecionadas para este servidor (*getClasses*). Com base nas informações requisitadas o objeto *gerarBusiness*, então efetua a agregação dos objetos através do método “*gerar*”.

FIGURA 21 DIAGRAMA DE SEQÜÊNCIA GERAR *BUSINESS SERVER*

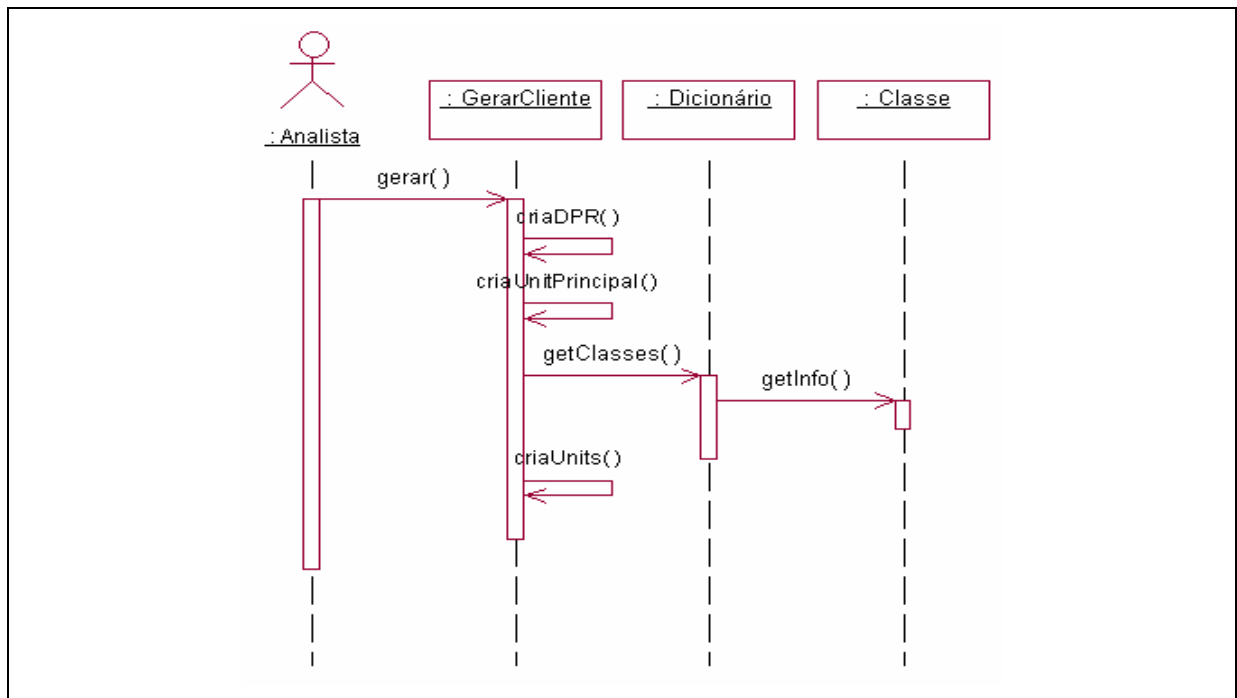


### 6.2.3.7 GERAR CLIENTE

Este diagrama de seqüência, identificado na Figura 22, tem o objetivo de gerar uma aplicação cliente para acessar as funcionalidades dos objetos de negócio. O início dá-se quando um ator que possui o papel de analista envia uma mensagem “*gerar*” para um objeto “*gerarCliente*”, então este objeto, responsável pela criação do código fonte do cliente cria seu programa principal, através da mensagem “*gerarDPR*” e sua *unit* principal através do método “*criaUnitPrincipal*”.

Finalmente é enviado uma mensagem para um objeto “dicionário” requisitando suas classes que possuirão uma *interface* com o usuário e para cada uma dessas classes é criado uma *unit* através da mensagem “*criaUnit*”.

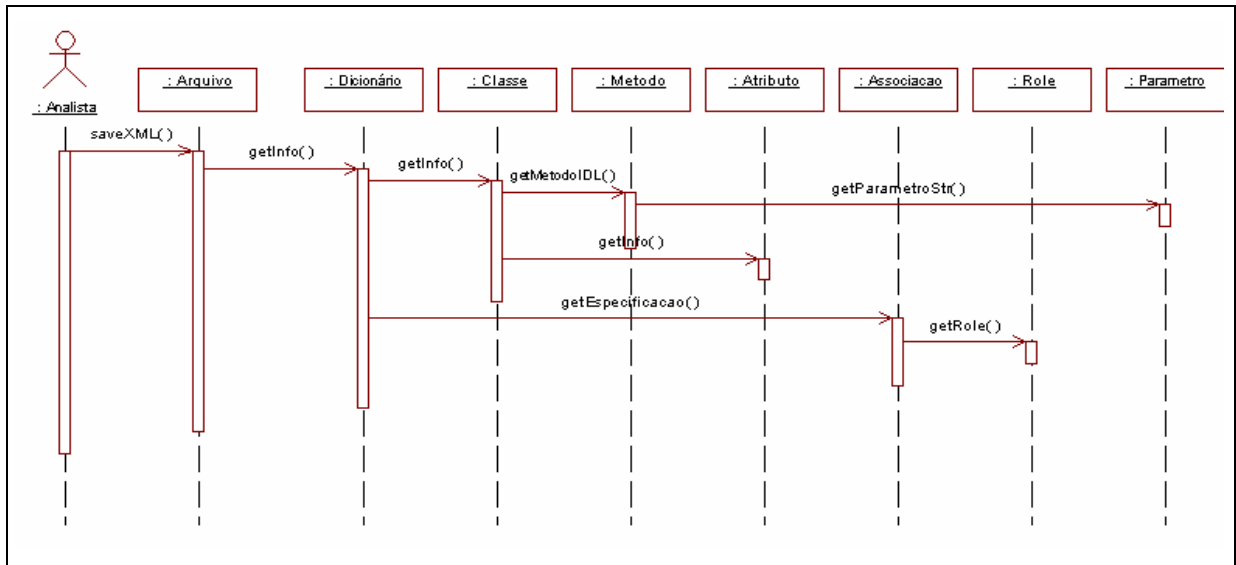
FIGURA 22 DIAGRAMA DE SEQÜÊNCIA GERAR CLIENTE



### 6.2.3.8 SALVAR XML

O diagrama de seqüência identificado na Figura 23, tem início com o envio de uma mensagem “salvarXML” pelo analista a um objeto “arquivo”. Então este objeto requisita a um objeto “dicionário” suas informações para que sejam armazenadas em um arquivo XML. O objeto “dicionário”, por sua vez requisita as informações sobre as classes para o objeto “classe”, “método”, “atributos” e “associação”. O objeto “método” ainda necessita de seus parâmetros que são obtidos através do envio da mensagem “getParametrosStr” a um objeto “parâmetro”. O objeto “associação” também precisa das informações referentes aos papéis da associação; para isso envia um mensagem “getRole” a seus objetos “role”.

FIGURA 23 SALVAR XML

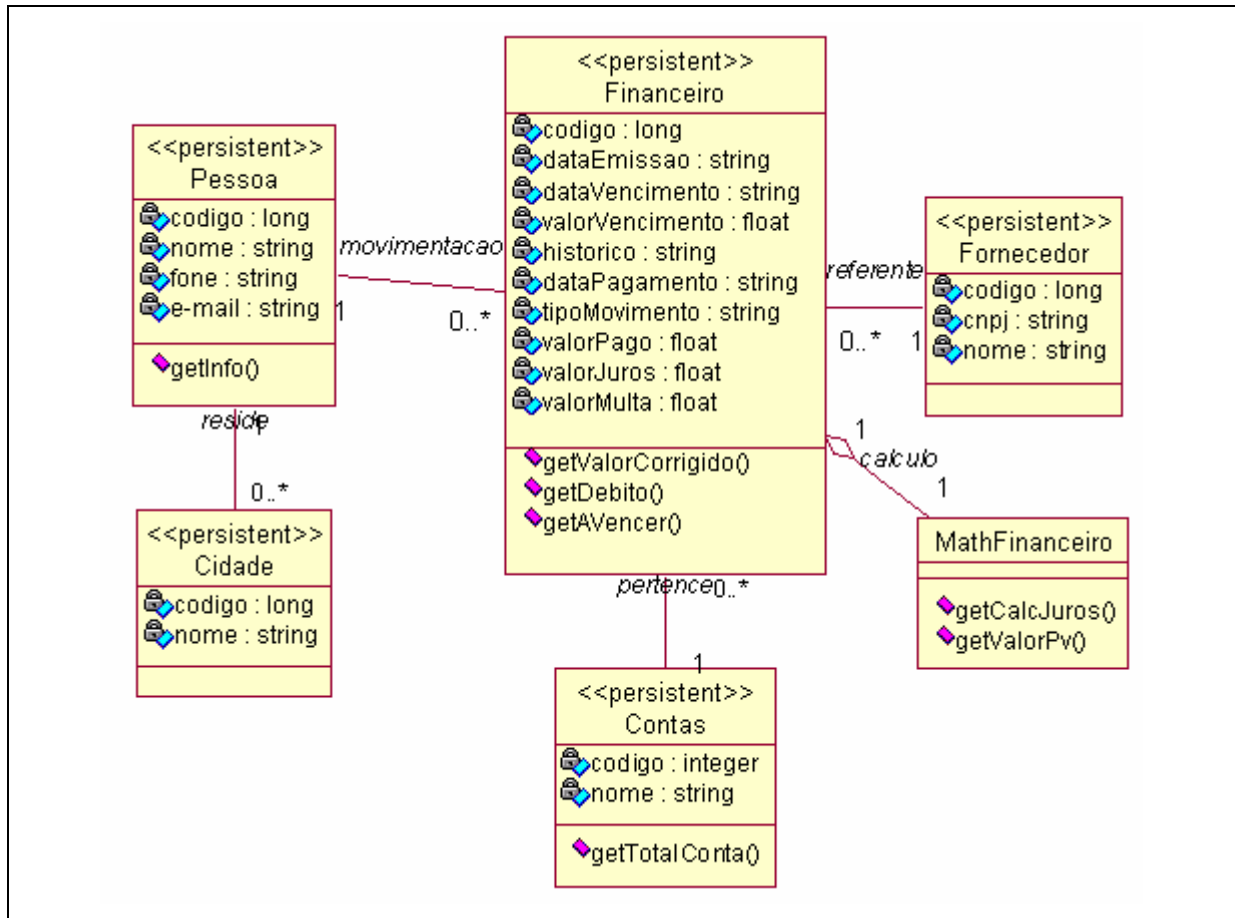


## 6.3 IMPLEMENTAÇÃO

O protótipo foi implementado no ambiente de programação Delphi 6, utilizando a biblioteca *Cross Platform* (CLX), para que a aplicação possa ser portátil para os sistemas operacionais Windows e Linux.

Para demonstrar a operacionalidade, será utilizado um estudo de caso, que tem o objetivo de controlar os pagamentos efetuados de uma pessoa. O diagrama de classes pode ser observado na Figura 24.

FIGURA 24 DIAGRAMA DE CLASSES DO ESTUDO DE CASO



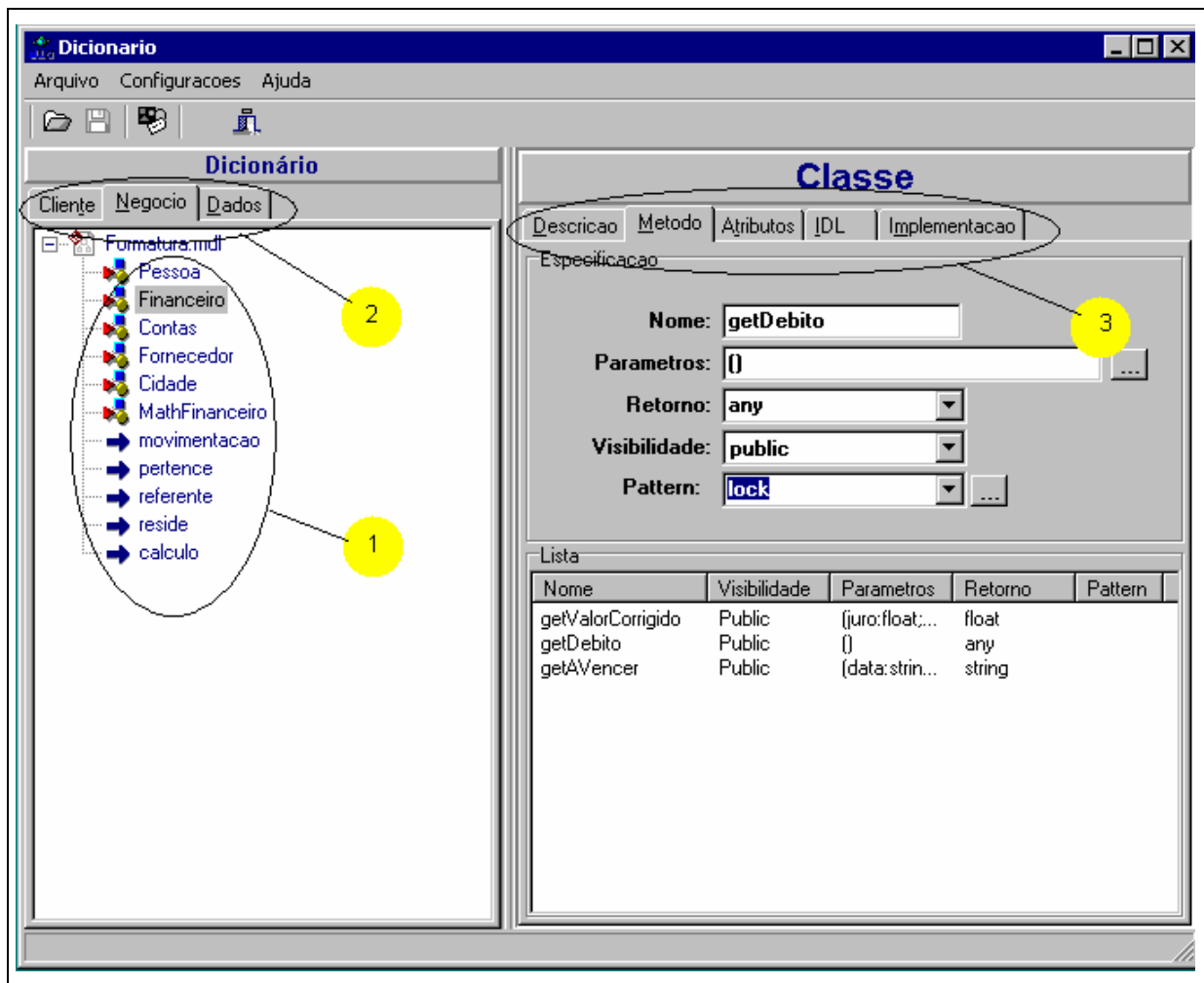
Para utilizar a ferramenta deve-se inicialmente criar o diagrama de classes, apresentado na Figura 24, na ferramenta CASE Rational Rose 2000.

Tendo o diagrama salvo, pode-se iniciar a geração do sistema distribuído, que está dividido em três etapas, apresentadas a seguir.

### 6.3.1 DICIONÁRIO

A Figura 25 apresenta a tela principal do protótipo, apresentando o diagrama de classes da aberto.

FIGURA 25 TELA PRINCIPAL DICIONÁRIO



As principais funções são descritas a seguir:

a) arquivo

- abrir arquivo do Rational Rose: abre um arquivo do Rational Rose 2000 e cria em memória as definições das classes, métodos, parâmetros, atributos e associações (Figura 25, item 1). Para cada arquivo lido é criado um nodo na árvore.
- salvar XML: salva a definição do diagrama de classes em XML.

b) configurações

- propriedades: apresenta algumas configurações possíveis ao dicionário como: caminho do compilador IDL e diretório das bibliotecas de *Design Patterns*. Este arquivo de configuração também é um arquivo XML.

A utilização da linguagem XML no dicionário tem as seguintes justificativas:

- a) portátil: esta aplicação pode ser compilada em Delphi 6 para ser utilizada no sistema operacional Windows ou compilada para ser executada em Linux utilizando o Kylix 2.;
- b) manipulação facilitada: o arquivo XML possui uma estrutura muito mais fácil de ser manipulada que um arquivo do Rational Rose (Quadro 18);
- c) facilidade de verificar a consistência de um arquivo: através da DTD, que possui as regras para validação do arquivo (Quadro 19);
- d) facilidade em gerar a documentação das classes, pois é possível definir folhas de estilo para sua apresentação, que pode ser através de arquivos HTML, PDF, etc.

#### QUADRO 18 ARQUIVO XML DO DICIONÁRIO

```

<?xml version="1.0" ?>
<!DOCTYPE dic.dtd (View Source for full doctype...)>
- <ListaDicionarios>
- <Dicionario Nome="Formatura.mdl">
+ <Classe Nome="Pessoa">
+ <Classe Nome="Financeiro">
- <Classe Nome="Contas">
  <Documentacao />
  <Descricao />
  <Visibilidade>public</Visibilidade>
- <Metodo Nome="getTotalConta">
  <Visibilidade>Public</Visibilidade>
  <Result>float</Result>
- <ListaParametros>
  - <Atributo Nome="conta">
    <Tipo>integer</Tipo>
    </Atributo>
  </ListaParametros>
</Metodo>
- <Atributo Nome="codigo">
  <Visibilidade>private</Visibilidade>
  <Tipo>integer</Tipo>
</Atributo>
- <Atributo Nome="nome">
  <Visibilidade>private</Visibilidade>
  <Tipo>string</Tipo>
</Atributo>
</Classe>
+ <Classe Nome="Aluno">
+ <Classe Nome="Fornecedor">
+ <Classe Nome="Cidade">
+ <Classe Nome="MathFinanceiro">
+ <Associacao Nome="movimentacao">
+ <Associacao Nome="pertence">
+ <Associacao Nome="referente">
+ <Associacao Nome="reside">
+ <Associacao Nome="calcula">
  <Descricao>obs</Descricao>
</Dicionario>
</ListaDicionarios>

```

### QUADRO 19 DTD DO SALVAR ARQUIVO EM XML

```

<!ELEMENT ListaDicionarios (Dicionario*) >
<!ELEMENT Dicionario (Classe*,Associacao ,Descricao ) >
  <!ATTLIST Dicionario Nome CDATA #IMPLIED >
<!ELEMENT Classe (Documentacao ,Descricao ,Visibilidade
,Metodo*,Atributo*,SuperClasse ) >
  <!ATTLIST Classe Nome CDATA #IMPLIED >
<!ELEMENT Documentacao (#PCDATA) >
<!ELEMENT Descricao (#PCDATA) >
<!ELEMENT Visibilidade (#PCDATA) >
<!ELEMENT Metodo (Visibilidade ,Result ,ListaParametros ) >
  <!ATTLIST Metodo Nome CDATA #IMPLIED >
<!ELEMENT Result (#PCDATA) >
<!ELEMENT ListaParametros (Atributo ) >
<!ELEMENT Atributo (Tipo ) >
  <!ATTLIST Atributo Nome CDATA #IMPLIED >
<!ELEMENT Tipo (#PCDATA) >
<!ELEMENT SuperClasse (#PCDATA) >
<!ELEMENT Associacao (Role*) >
  <!ATTLIST Associacao Nome CDATA #IMPLIED >
<!ELEMENT Role (Classe ,Agregacao ) >
  <!ATTLIST Role Nome CDATA #IMPLIED >
<!ELEMENT Agregacao (#PCDATA) >

```

As propriedades com relação aos métodos, atributos entre outros podem ser configurados no dicionário (Figura 25, item 2).

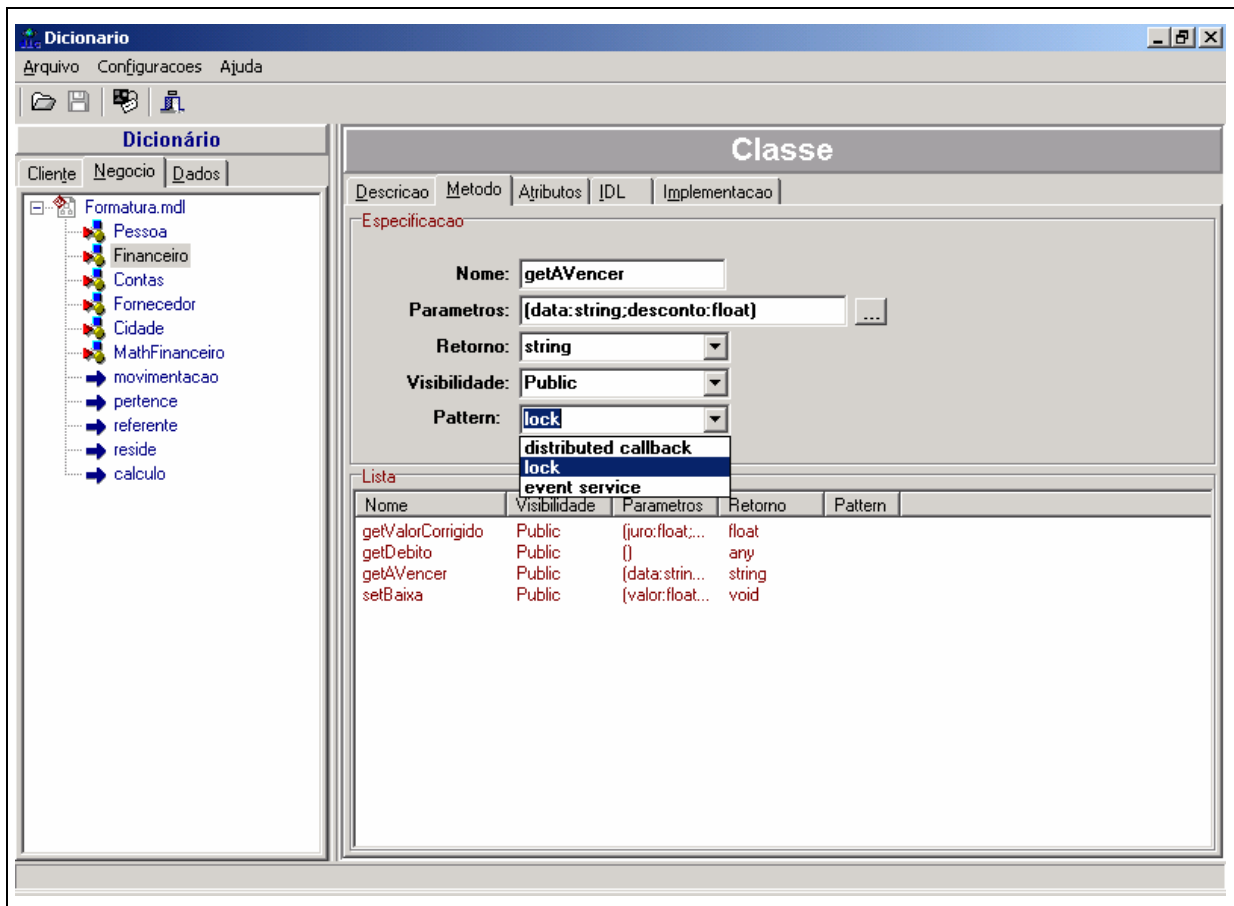
### 6.3.2 DESIGN PATTERNS

Após ter carregado as definições do arquivo em memória o protótipo disponibiliza as classes em três categorias diferentes: cliente, negócio e dados (Figura 25, item 3). A categoria cliente contém todas as classes que irão possuir uma camada de aplicação cliente, ou seja, possuirá uma *interface* com o usuário final. Inicialmente todas as classes que possuem o estereótipo *persistent* estarão nesta categoria. Esta camada possui os padrões *persistentLayer* e o padrão *template* para geração do código fonte do programa cliente. A categoria negócio possui todas as classes do diagrama de classes que foi lido. Nesta categoria é possível aplicar os padrões *lock*, *distributed Callback*, *event service* e implementar o cliente do padrão *persistent layer* ser for o caso. A categoria dados conterà todas as classes que terão potencialmente um mapeamento para um banco de dados relacional. Esta camada implementa o padrão *persistent layer* e na aplicação de acesso a banco de dados implementa o padrão *factory*.

Os padrões podem ser definidos ao se fazer o diagrama de classes, definindo um estereótipo para uma classe, sendo que o único estereótipo visto como padrão é o estereótipo

*persistent*. Outra forma é definir no próprio protótipo, onde estes padrões podem ser aplicados a classes (*persistent layer*) ou a métodos (*lock, distributed Callback, event service*), como pode ser visto na Figura 26.

FIGURA 26 APLICAR *DESIGN PATTERN*



As classes onde os padrões foram aplicados são descrito a seguir:

- peessoa: utiliza o padrão *persistent layer*, pois é necessário armazenar seus dados e o método `getInfo` utiliza o padrão *distributed callback*;
- financeiro: também utiliza o padrão *persistent layer* e os métodos (`getValorCorrigido`, `getDebito` e `getAVencer`) possuem o padrão *lock* para que dois processo não acessem estes métodos ao mesmo tempo e ocorra inconsistência no resultado da operação. O método `setBaixa` utiliza o padrão de notificação para que cada vez que um documento é baixado todos os objetos tenham conhecimento da baixa.



- c) contas: utiliza o padrão *persistent layer*;
- d) fornecedor: utiliza o padrão *persistent layer*;
- e) cidade: utiliza o padrão *persistent layer*;
- f) mathFinanceiro: utiliza o padrão *distributed callback* nos métodos `getCalcJuros` e `getValorPV`, para que, enquanto este objeto processe seus cálculos, os objetos clientes possam continuar executando suas tarefas.

Após estas configurações é possível então gerar a interface IDL para um determinado objeto. A IDL incorpora algumas funcionalidades dependendo do padrão que é utilizado. Por exemplo, se for escolhido o padrão *distributed callback* o protótipo gera duas IDL's uma para implementação do servidor (Figura 27) e uma para o cliente (Figura 28).

FIGURA 27 IDL GERADA PARA O SERVIDOR

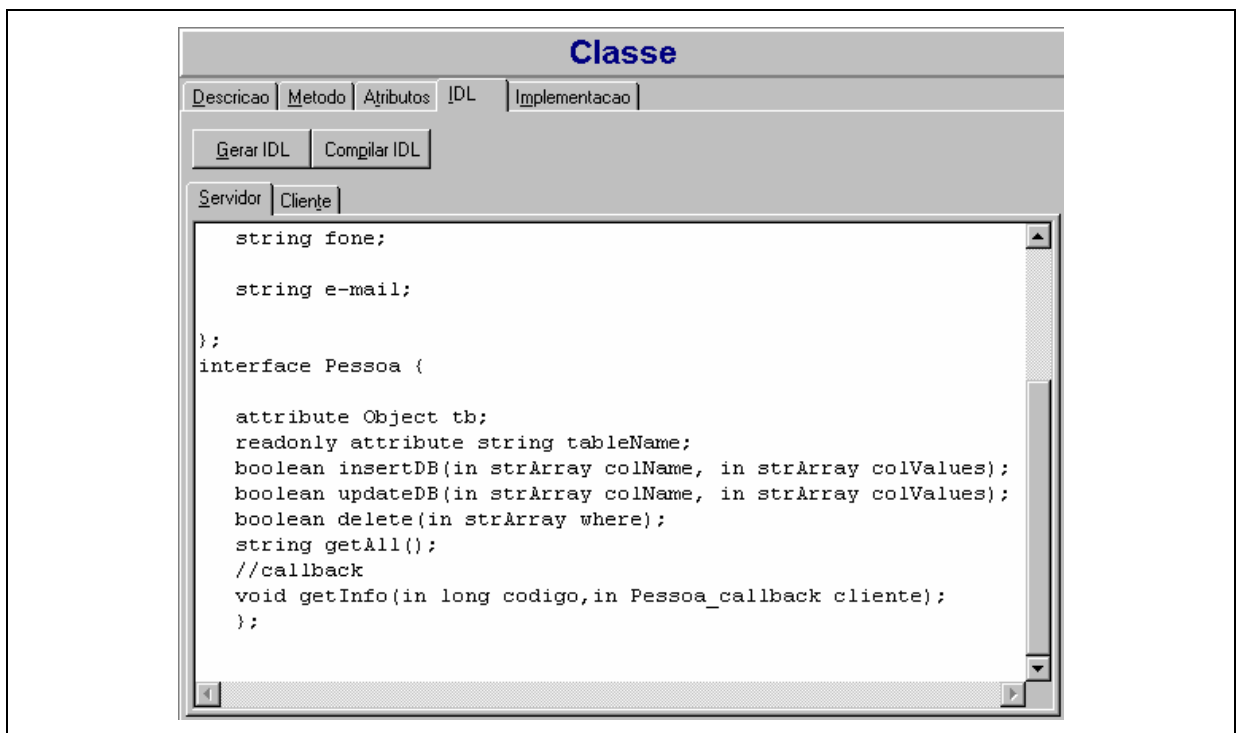
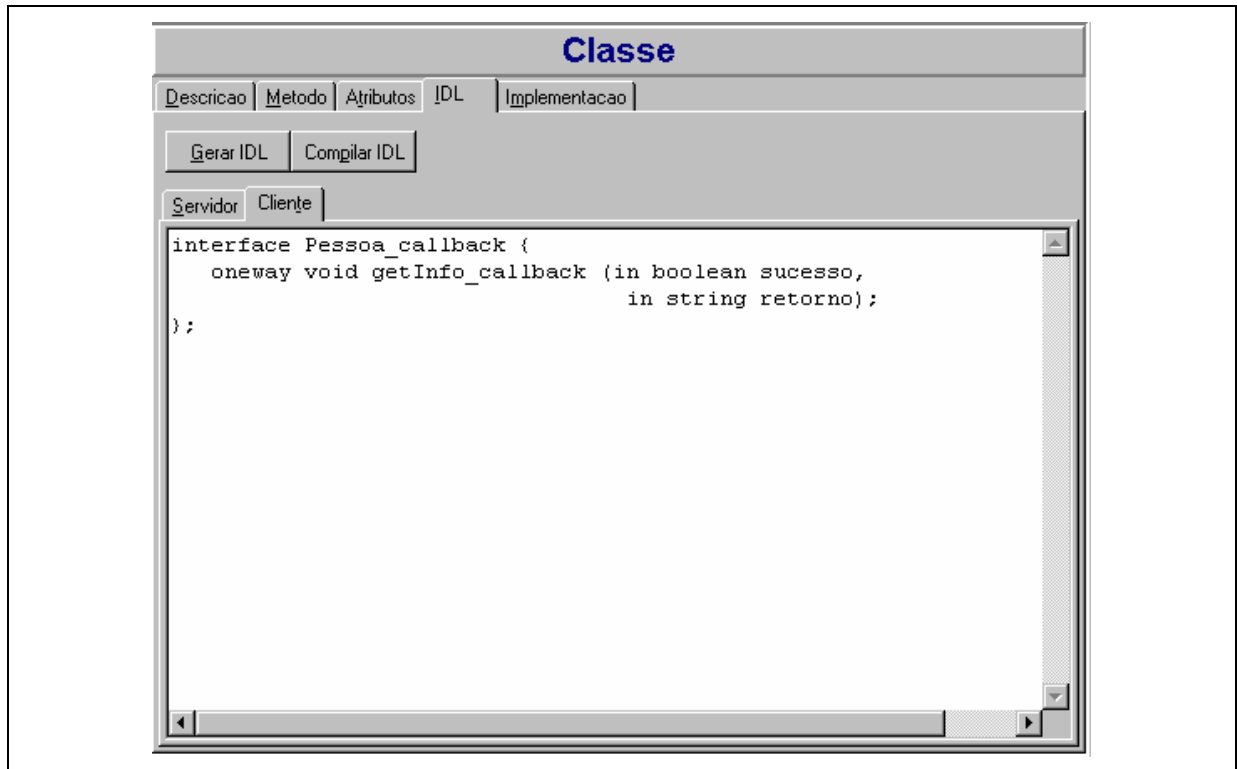
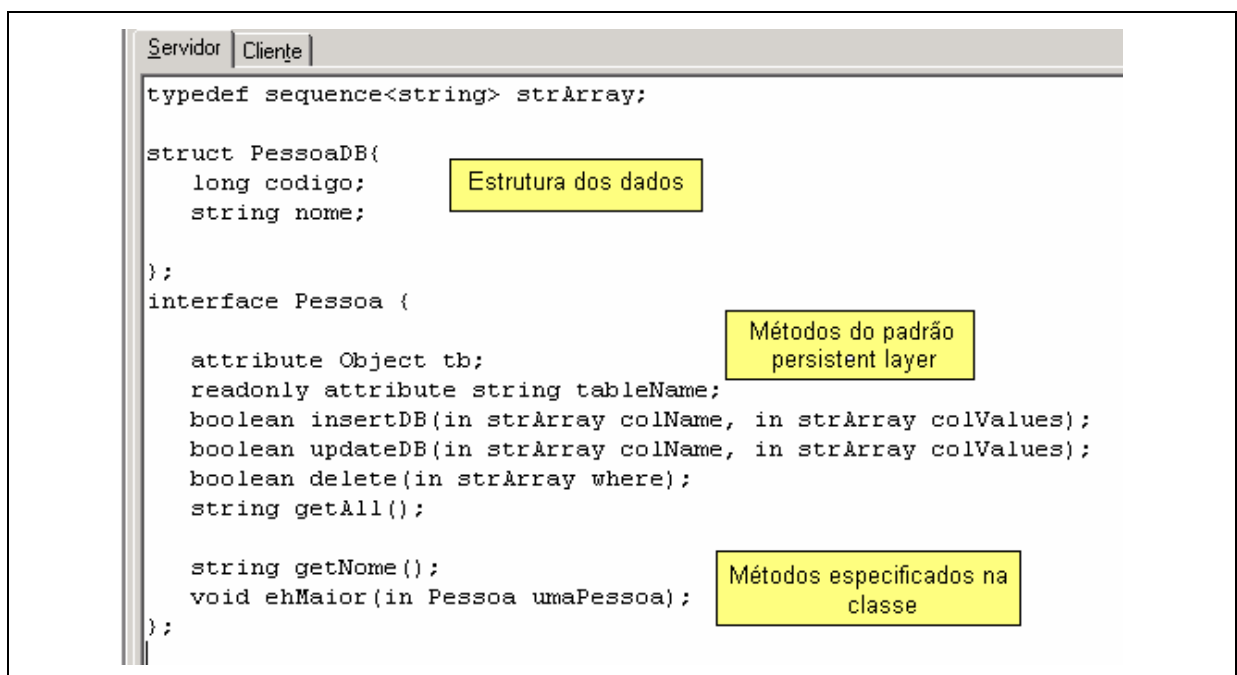


FIGURA 28 IDL GERADA PARA O CLIENTE



Se for aplicada à classe o padrão *persistent layer* a classe também possuirá a interface para manipulação e acesso aos dados, como pode ser visto na Figura 29.

FIGURA 29 IDL GERADA COM O PADRÃO *PERSISTENT LAYER*

Depois de gerada a IDL é necessário escolher a opção “compilar IDL” para executar o compilador IDL (idl2pas). Com o mapeamento para *object pascal* realizado pelo compilador, o protótipo insere as funcionalidades dependendo do padrão escolhido.

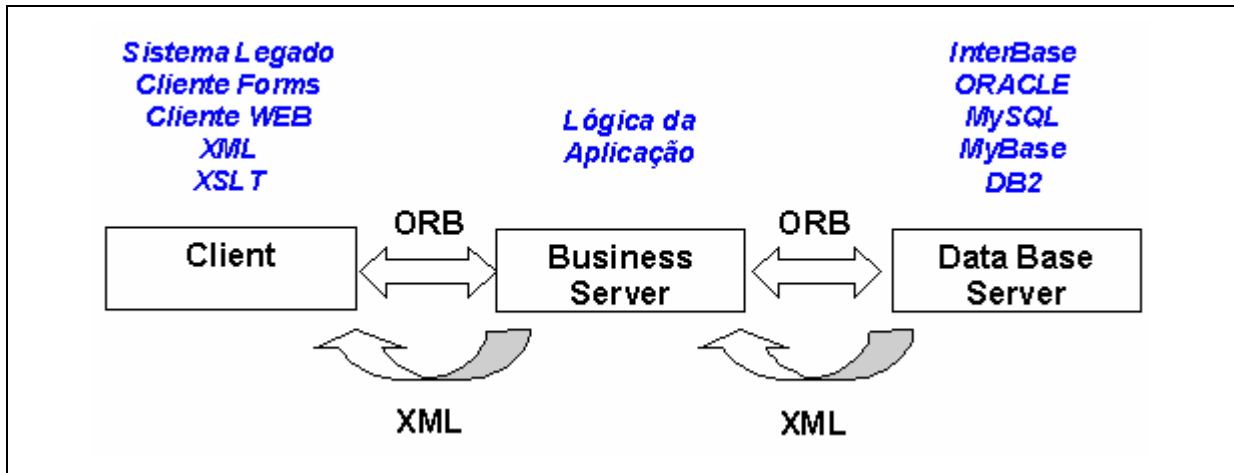
### 6.3.3 GERAR SISTEMAS DISTRIBUÍDOS

Finalmente, a última etapa do desenvolvimento é a geração dos sistemas distribuídos. Estes sistemas utilizam a tecnologia multicamadas (*multi-tier*), que é a divisão da aplicação camadas:

- a) *Client*: nesta camada são implementados recursos para prover uma *interface* com o usuário. Neste trabalho a *interface* com o usuário é uma aplicação WEB.
- b) *Business Server*: nesta camada contém a lógica da aplicação. Nela são implementadas as funcionalidades referentes à definição dos problemas realmente, como a definição do cálculo de imposto sobre uma determinada nota fiscal;
- c) *DataBase Server*: a camada de acesso a dados constitui da lógica para manipulação de um banco de dados relacional, como conexão, iniciar transações (*startTransaction*), confirmá-las (*commit*) ou retornar ao estado inicial da transação (*rollback*), executar *storage procedures*, *triggers*, entre outros recursos referentes ao acesso a um banco de dados relacional.

Esta arquitetura pode ser observada na Figura 30. É importante observar que os objetos que compõem as camadas da aplicação não necessitam estar na mesma máquina, pois eles podem estar distribuídos em uma rede de computadores. Podendo ocorrer que os objetos da camada *business server*, por exemplo, estejam também em máquinas diferentes. O mesmo pode ocorrer nas outras camadas da aplicação. Um mesmo objeto também pode estar replicado, isso garantindo a disponibilidade do sistema mesmo que ocorra algum problema em uma das máquinas da rede de computadores.

FIGURA 30 ARQUITETURA DE SISTEMAS DISTRIBUÍDOS

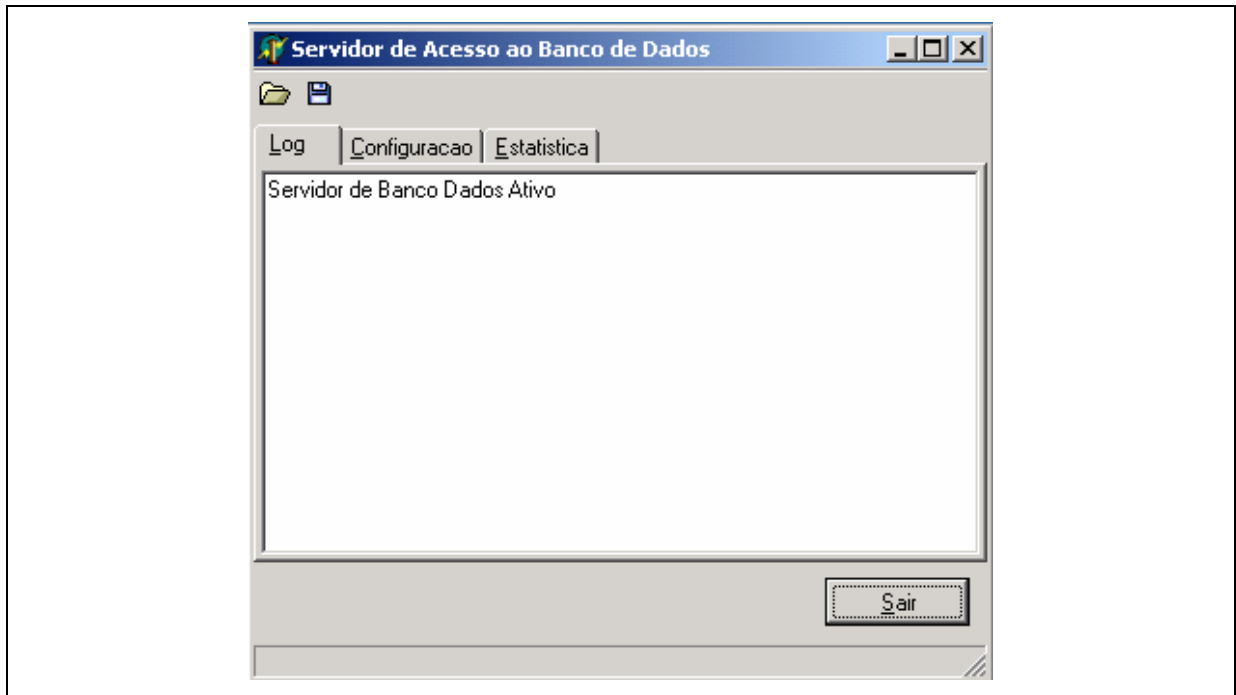


Os objetos da aplicação interagem graças ao ORB. Outro fator importante é que, quando alguma operação que envolva um grande número de dados de retorno é requisitada, como, por exemplo, quando é necessário apresentar todos os clientes que possuam alguma conta em débito, esta transferência da informação é feita através da XML. Isto o que garante que os diversos objetos distribuídos, que podem até estar implementados em linguagens diferentes, possam interagir de maneira facilitada, pois XML é uma linguagem padrão. Desta forma não é necessário que o cliente faça diversas requisições ao objeto servidor, o que aconteceria se ele estivesse utilizando apenas o ORB (seria necessário uma requisição para cada registro que fosse requisitado). E mais, o uso da XML possibilita que o cliente da aplicação seja uma aplicação com formulários, uma aplicação WEB, uma outra aplicação que utiliza o conceito de *WEBServices*, um documento XSLT, um sistema legado entre outros.

### 6.3.3.1 DATA BASE SERVER

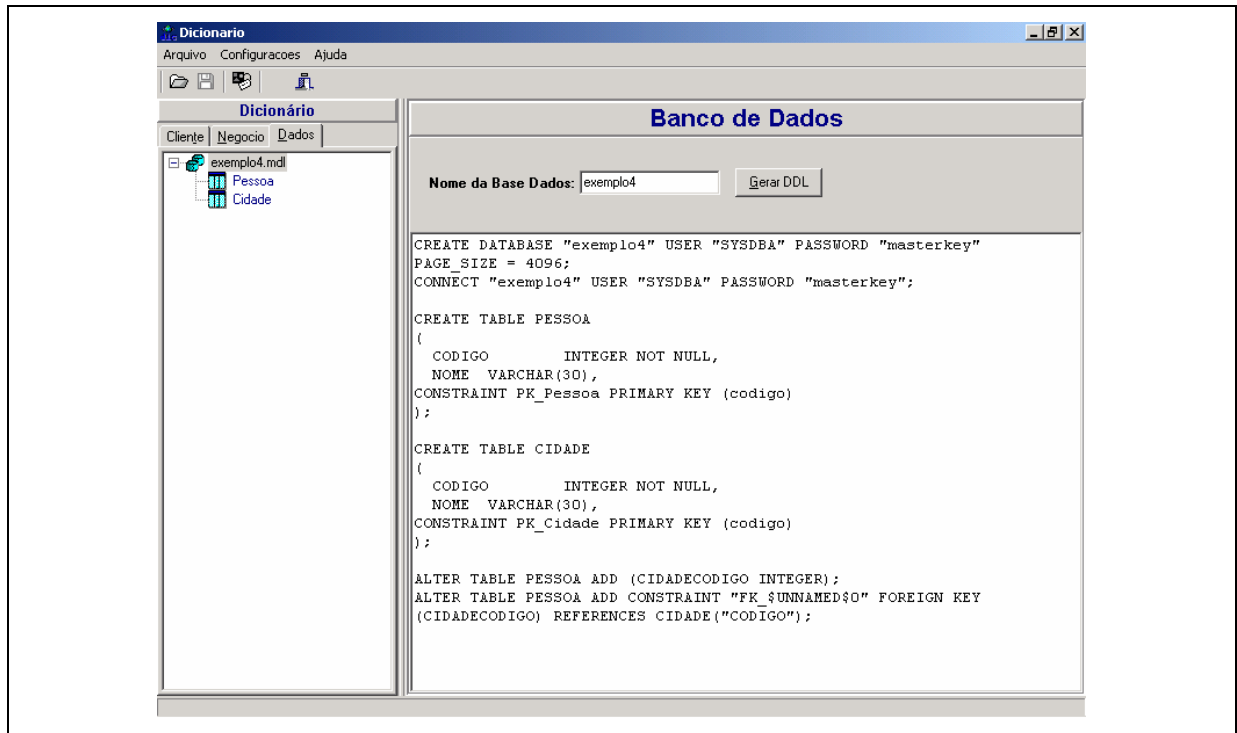
O servidor de acesso a dados segue as especificações do padrão *persistent layer*. O *DataBase Server* tem a funcionalidade de prover o acesso a um banco de dados relacional. Este protótipo está limitado aos banco de dados: Interbase, Oracle, Db2, MySql e MyBase, devido à disponibilidade de *drivers* de acesso à banco de dados para os componentes *DBExpress*, que são os responsáveis pelo acesso ao banco de dados quando utilizadas as bibliotecas CLX. A Figura 31, mostra o *DataBase Server* executando.

FIGURA 31 DATABASE SERVER



O protótipo gera o *script* para criação do banco de dados com base nas classes que possuem o padrão *persistent*, como pode ser observado na Figura 32.

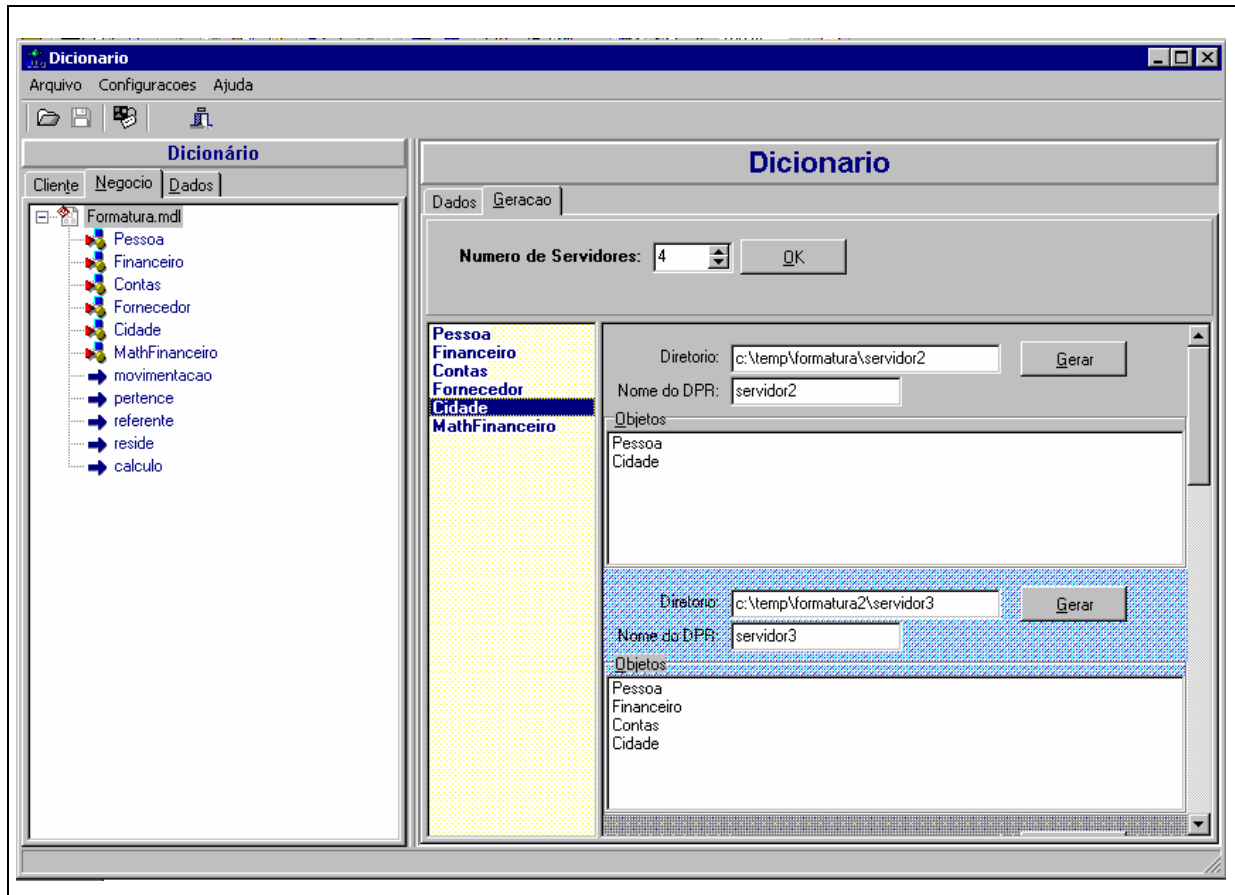
FIGURA 32 GERAR DDL



### 6.3.3.2 BUSINESS SERVER

Os servidores de objetos de negócio são gerados com base nas informações contidas no dicionário. Um objeto pode ser replicado em vários servidores de objetos. A finalidade de possuir diversos servidores de objetos é o aumento da escalabilidade do sistema. Para gerar os servidores de negócio, primeiramente é escolhido o número de servidores; após é escolhido o diretório em que o servidor será criado, o nome do programa principal e os objetos que compõem o servidor. Então deve-se executar a geração, acionando o botão “gerar”, como pode ser observado na Figura 33.

FIGURA 33 TELA DO GERAR BUSINESS SERVER



O protótipo gera o programa principal, como pode ser observado no Quadro 20.

QUADRO 20 PROGRAMA PRINCIPAL GERADO

```

Program BusinessServer;

uses
  QForms,
  UBusinessServer in 'c:\tcc\lib\UBusinessServer.pas' {FrmBusinessServer},
  uPrincipal in 'uPrincipal.pas' {FrmPrincipal},
  persistentLayer_i in 'c:\tcc\lib\persistentLayer_i.pas',
  persistentLayer_c in 'c:\tcc\lib\persistentLayer_c.pas',
  Pessoa_s in 'c:\temp\Pessoa_s.pas',
  Pessoa_i in 'c:\temp\Pessoa_i.pas',
  Pessoa_impl in 'c:\temp\Pessoa_impl.pas',
  Pessoa_c in 'c:\temp\Pessoa_c.pas',
  PushSupplier_Impl in 'c:\tcc\lib\PushSupplier_Impl.pas',
  Financeiro_s in 'c:\temp\Financeiro_s.pas',
  Financeiro_i in 'c:\temp\Financeiro_i.pas',
  Financeiro_impl in 'c:\temp\Financeiro_impl.pas',
  Financeiro_c in 'c:\temp\Financeiro_c.pas';

```

```

Begin
  Application.Initialize;
  Application.CreateForm(TFrmBusiness, FrmBusiness);
  Application.Run;
end.

```

Também é gerada a *unit* principal que contém a interface do servidor, bem como as declarações dos objetos e ligações com o servidor de acesso a dados se for o caso. A *unit* principal pode ser observada no Quadro 21.

#### QUADRO 21 UNIT PRINCIPAL GERADA

```

unit uPrincipal;

interface

uses
  corba,
  persistentLayer_c, persistentLayer_i, // acesso ao DataBaseServer
  { objetos de negócio }
  Pessoa_s,
  Pessoa_i,
  Pessoa_impl,
  Pessoa_c,
  Financeiro_s,
  Financeiro_i,
  Financeiro_impl,
  Financeiro_c,
  CosEvent, PushSupplier_Impl, // eventos
  { interface }
  SysUtils, Types, Classes, Qgraphics, QControls, QForms, QDialogs,
  QStdCtrls, uBusinessServer, QcomCtrls, QButtons, QExtCtrls;

Type
  TFrmBusiness = class(TFrmBusinessServer)
    Procedure FormCreate(Sender: TObject);
  Private
    { Private declarations }
  protected
  tb : TableManager; // acesso ao DataBaseServer

    {declaração dos objetos }
    fPessoa : Pessoa;
    fFinanceiro : Financeiro;
  public
    {declaração do canal de eventos }
    Event_Channel : EventChannel;
    Supplier_Admin : SupplierAdmin;
    Push_Consumer : ProxyPushConsumer;
    PushSupplier_Skeleton : PushSupplier;

    { Public declarations }
  end;

```



```

var
  FrmBusiness: TFrmBusiness;

Implementation

{$R *.xfrm}

procedure TFrmBusiness.FormCreate(Sender: TObject);
begin
  inherited;
  {Ligar com o DataBaseServer }
  tb:= TtableManagerHelper.Bind();
  {Pessoa}
  fPessoa := TPessoaSkeleton.Create('Pessoa Server ', TPessoa.Create);
  fPessoa.tb := CorbaObject(tb);
  BOA.ObjIsReady(fPessoa as _Object);
  ListBoxLog.items.add('Pessoa está ativo...');
  {Financeiro}
  fFinanceiro := TFinanceiroSkeleton.Create('Financeiro Server ',
TFinanceiro.Create);
  fFinanceiro.tb := CorbaObject(tb);
  BOA.ObjIsReady(fFinanceiro as _Object);
  ListBoxLog.items.add('Financeiro está ativo...');

  // cria um skeleton e registra com Basic Object Adapter
  PushSupplier_Skeleton := TPushSupplierSkeleton.Create('Evento ',
TPushSupplier.Create);
  BOA.SetScope( RegistrationScope(1) );
  BOA.ObjIsReady(PushSupplier_Skeleton as _Object);

  //bind o canal do evento e pega o objeto servidor
  Event_Channel := TEventChannelHelper.bind;
  Supplier_Admin := Event_Channel.for_suppliers;

  //pega um push cliente e registra o objeto servidor
  Push_Consumer := Supplier_Admin.obtain_push_consumer;
  Push_Consumer.connect_push_supplier(PushSupplier_Skeleton);

  //
end;

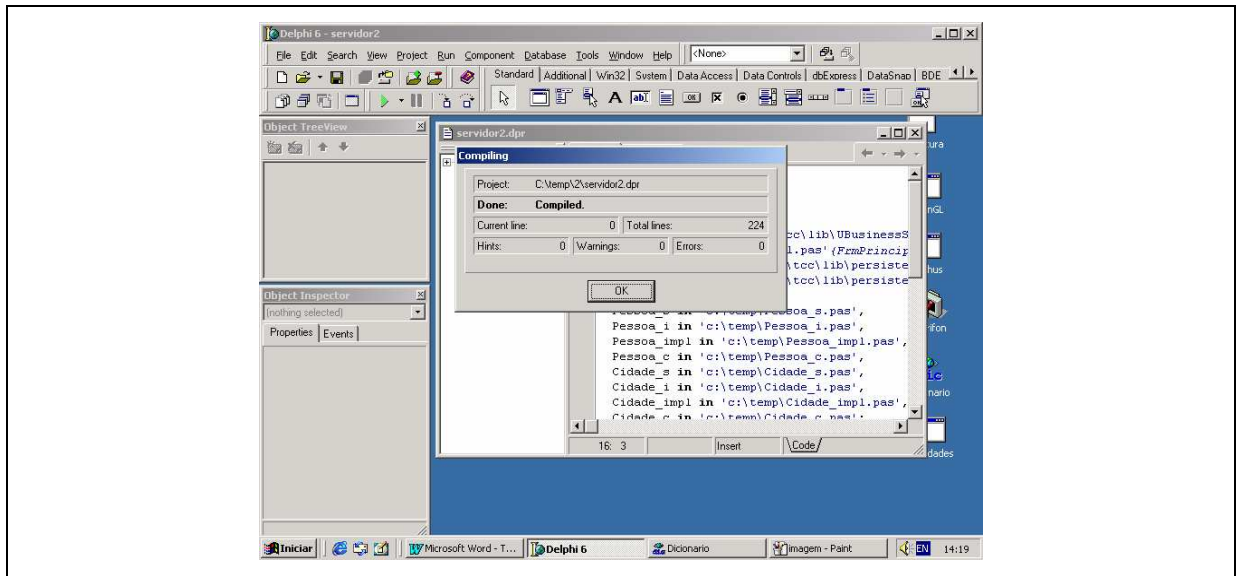
end.

```

Finalmente é gerada a implementação do objeto servidor com os *Design Patterns* selecionados, como pode ser observado no Anexo 1.

Depois de gerado o código fonte dos programas, basta compilá-los, como pode ser observado na Figura 34.

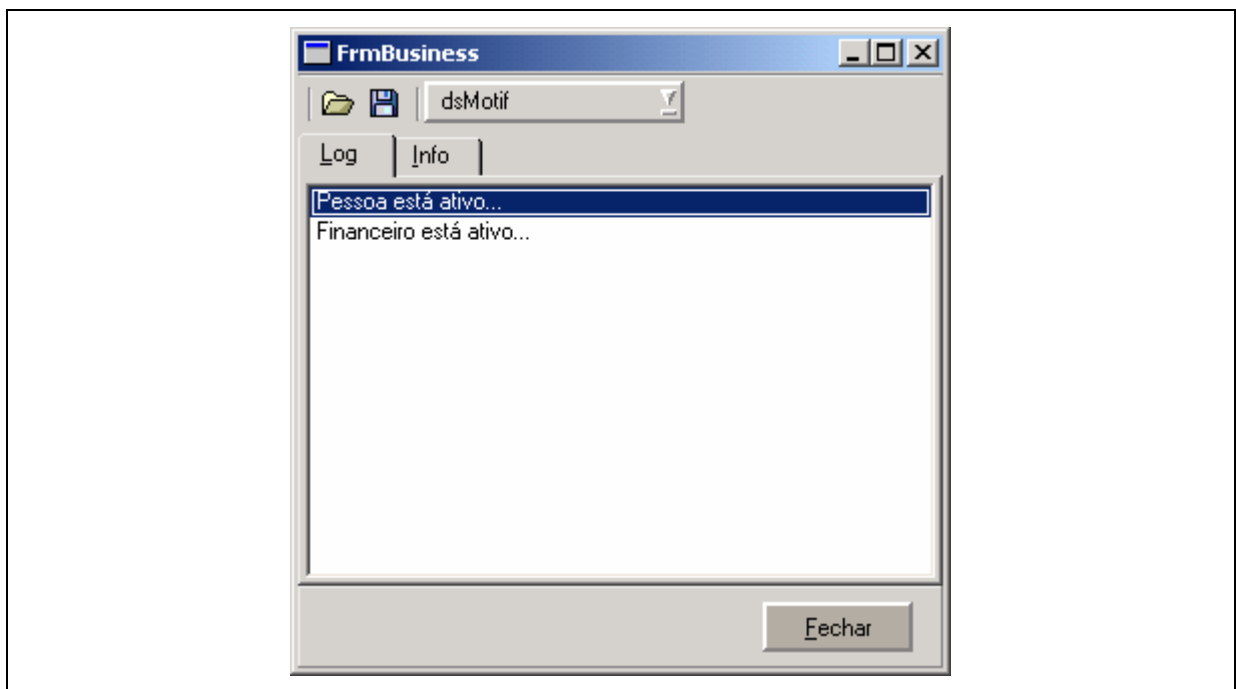
FIGURA 34 COMPILANDO O SERVIDOR DE NEGÓCIOS



O protótipo gera as funcionalidades básicas e os padrões quando selecionados. Todas as implementações restantes como a lógica do negócio devem ser adicionadas ao projeto.

A Figura 35 mostra o servidor de negócios executando.

FIGURA 35 SERVIDOR DE NEGÓCIO EXECUTANDO

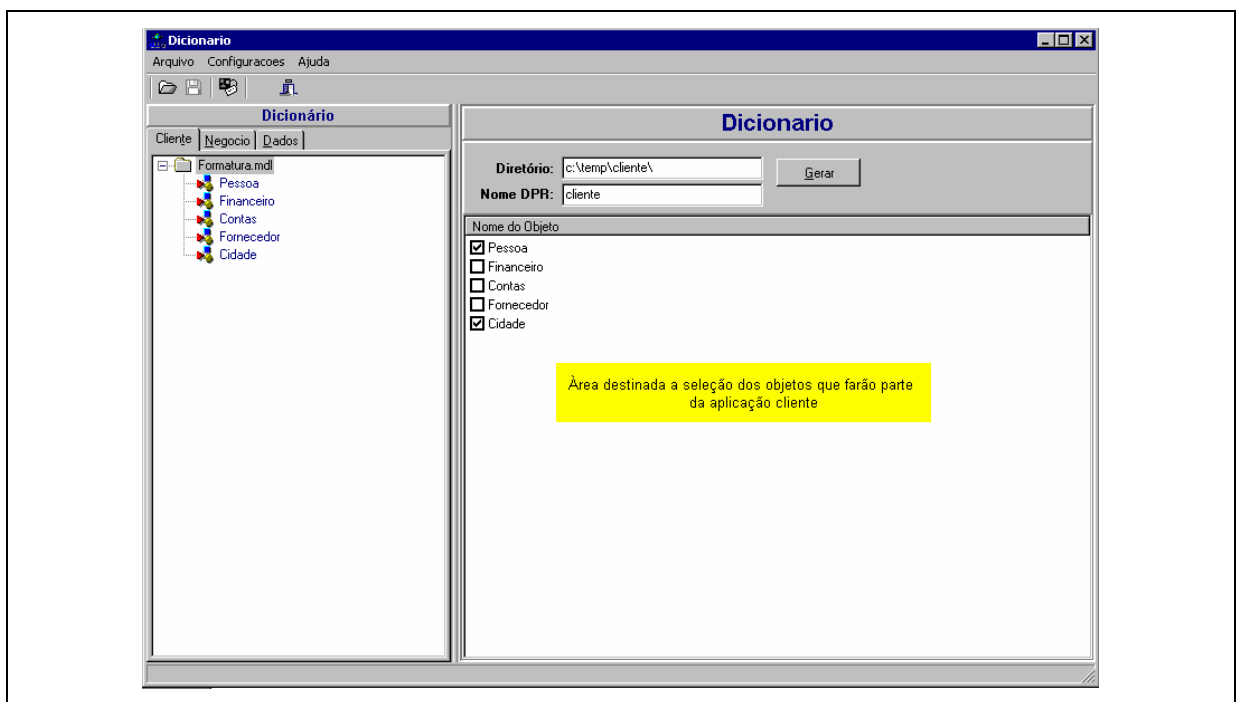


### 6.3.3.3 CLIENTE

O cliente define uma interface para que o usuário possa interagir com o sistema distribuído.

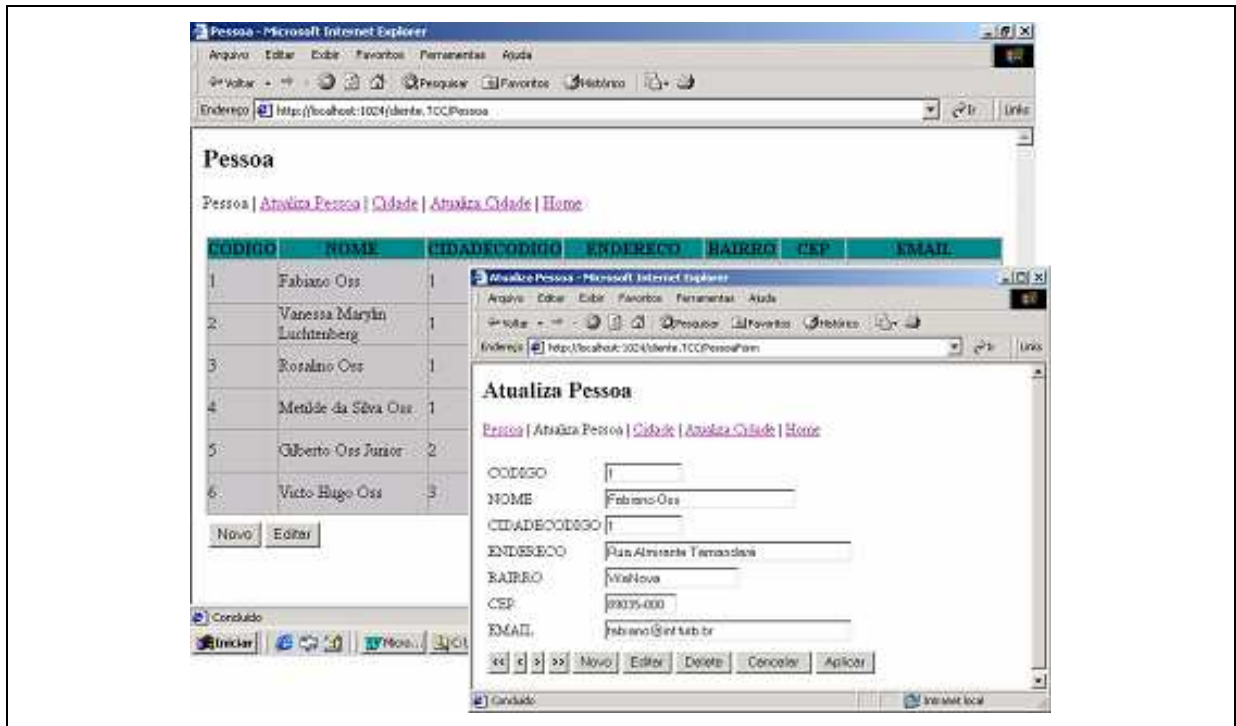
A Figura 36, mostra como é gerada uma aplicação cliente. Primeiramente é necessário definir o nome do diretório e o nome do projeto para gerar a aplicação. Após é escolhido quais objetos possuirão uma interface WEB, então basta executar o botão “gerar” para o protótipo criar a aplicação cliente.

FIGURA 36 GERANDO UMA APLICAÇÃO CLIENTE



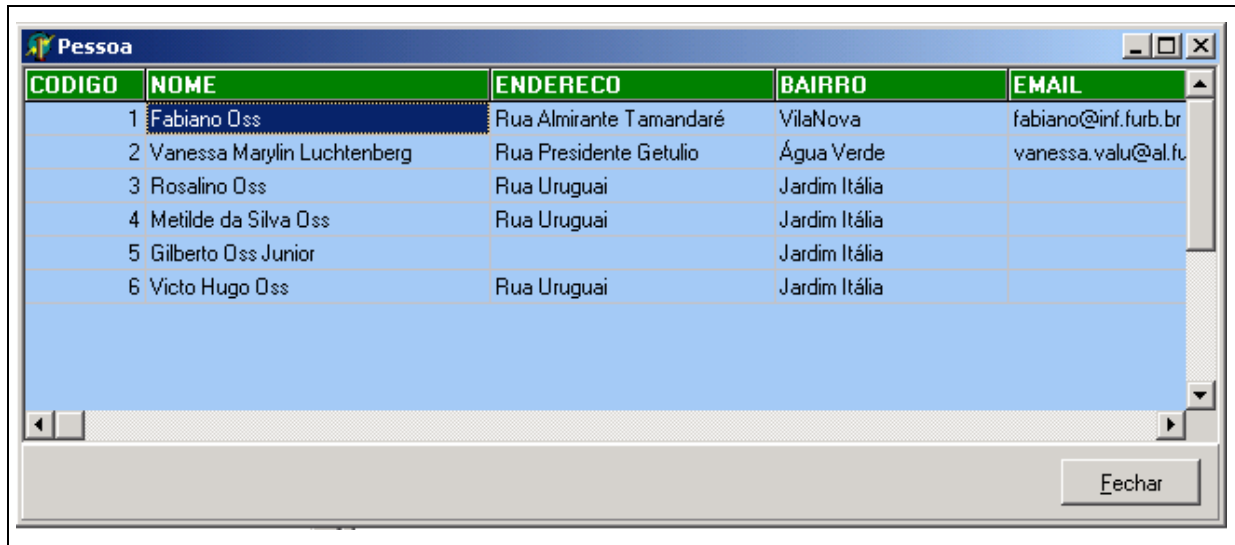
O protótipo gera a aplicação cliente baseado em XML, como pode ser observado na Figura 37.

FIGURA 37 INTERFACE HTML



O programa principal da aplicação cliente pode ser observado no Anexo 2 e a implementação de uma interface pode ser visto no Anexo 3. Para geração da interface HTML é utilizado o padrão *template*.

Se for necessária a criação uma interface com formulários, não há necessidade de alterar nada nos outros objetos, apenas é necessário implementar o cliente como pode ser observado na Figura 38.

FIGURA 38 *INTERFACE COM FORMULÁRIOS*

CODIGO	NOME	ENDERECO	BAIRRO	EMAIL
1	Fabiano Oss	Rua Almirante Tamandaré	VilaNova	fabiano@inf.furb.br
2	Vanessa Marylin Luchtenberg	Rua Presidente Getulio	Água Verde	vanessa.valu@al.fu
3	Rosalino Oss	Rua Uruguai	Jardim Itália	
4	Metilde da Silva Oss	Rua Uruguai	Jardim Itália	
5	Gilberto Oss Junior		Jardim Itália	
6	Victo Hugo Oss	Rua Uruguai	Jardim Itália	

Fechar

## 7 CONCLUSÕES

Com este trabalho foi possível verificar que as aplicações baseadas em sistemas distribuídos estão surgindo para resolver os desafios atuais da área da tecnologia da informação como acessibilidade à internet, integração entre as aplicações, disponibilidade dos sistemas e escalabilidade. Dentre todas as tecnologias de objetos distribuídos o padrão CORBA destaca-se pela sua heterogeneidade.

Sobre os *Design Patterns* verificou-se que ajudam a reduzir a complexidade e tempo para o desenvolvimento das aplicações e promovem a reutilização dos esforços de desenvolvimento. A utilização de padrões também aumenta a confiabilidade da aplicação uma vez que utiliza padrões que já foram testados.

O padrão de persistência de dados mostrou-se eficiente na manipulação de banco de dados relacional, embora possa melhorar incorporando a buferização de dados e o padrão de notificação. Os demais padrões (*callback*, *lock*, notificação, *factory e template*), tiveram um bom desempenho mostrando-se eficazes no seu contexto.

Sobre a ferramenta CASE Rational Rose verificou-se que ela tem um bom suporte a linguagem UML e permitiu a interface com a ferramenta gerada.

A arquitetura de sistemas distribuídos apresentada, utilizando CORBA com XML, mostra-se eficiente na transferência de informação entre as camadas, possuindo grande interoperabilidade entre os objetos, evidenciando a padronização e a independência de plataforma embora tenha um tempo de resposta maior devido à conversão dos dados que estão em uma base de dados relacional para XML.

Quanto ao ambiente de desenvolvimento Delphi 6.0, apesar de ter melhorado o suporte a CORBA desde a versão 5.0, ainda existem muitas características que ele não suporta como: serviço de segurança, serviço de controle de concorrência, a utilização de outro ORB, entre outras. O Delphi é uma boa escolha para o desenvolvimento da aplicação cliente, já para aplicação servidor a utilização de Java ou C++ seria mais indicada. As bibliotecas CLX mostraram algumas falhas principalmente no que se refere ao alinhamento dos componentes na tela e a falta de muitas propriedades comparado com as bibliotecas VCL. Como aspecto positivo pode-se citar a portabilidade em relação ao sistema operacional.

O sistema gerado atendeu as operacionalidade descritas. A compilação em Delphi 6 e Kylix 2 foi realizada com sucesso embora a aplicação possua alguns problemas no sistema operacional Linux e apresenta um bom funcionamento no sistema operacional Windows.

O *middleware* Visibroker, mostrou um bom desempenho para esta aplicação, e facilidades em gerenciar os objetos através de seu aplicativo console.

## 7.1 PROBLEMAS E LIMITAÇÕES

Como problemas e limitações deste trabalho pode-se destacar:

- a) a leitura do diagrama de classes está limitado ao Rational Rose 2000;
- b) só é gerado os sistemas distribuídos para linguagem *object pascal*;
- c) os sistemas distribuídos gerados e compilados com o Kylix 2, não executaram normalmente devido a problemas na biblioteca *liborpas45.so.1*.

## 7.2 EXTENSÕES

Como sugestões para continuação deste trabalho destacam-se:

- a) tornar a geração dos sistemas genérico para abranger outra linguagem de programação;
- b) definir novos *Design Patterns*;
- c) desenvolver padrões para outras arquiteturas de sistemas distribuídos como COM, *Enterprise Java Beans*, *SOAP*, entre outras. O *SOAP* é um protocolo definido em XML, sendo assim, as chamadas a procedures remotas são codificadas em XML. Para transporte das mensagens é usado o HTTP, que além de tornar o SOAP um protocolo leve, elimina inúmeros problemas de outras tecnologias com proxys, como CORBA e DCOM;
- d) implementar a leitura do diagrama de classes para outra ferramenta CASE como *Togheter*, *Objectering*, entre outras;
- e) tornar os padrões independentes de linguagem criando uma forma de especificação dos padrões como por exemplo XML ou BNF;
- f) implementar um tradutor de Rational Rose para XML independente, para que o diagrama de classes possa ser reutilizado em outras aplicações;
- g) aumentar a poder de configuração para a geração dos objetos distribuídos.

## REFERÊNCIAS BIBLIOGRÁFICAS

BOOCH, Grady, RUMBAUGH, James, JACOBSEN, Ivar. **UML guia do usuário**. Tradução Fábio Freitas da Silva. Rio de Janeiro: Campus, 2000.

BOSIO, Rubens. **Análise comparativa entre as especificações de objetos distribuídos DCOM e CORBA utilizando o ambiente de desenvolvimento Delphi**. 2000. 55 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

BROSE, Gerald, VOGEL, Andreas, DUDDY, Keith. **Java programming with CORBA – advanced techniques for building distributed applications**. New York: John Wiley, 3ed, 2001.

CAGNIN, Maria Istela et al. Reengenharia com Uso de Padrões de Projeto. In: XIII Simpósio Brasileiro de Engenharia de Software, 1., 1999, Florianópolis. **Anais...** Florianópolis: UFSC, 1999. p. 273-289.

CANTÚ, Marco. **Dominando o Delphi 5 – a bíblia**. Tradução João E.N. Tortello. São Paulo: Markon Books, 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, NORTH, David, MAYFIELD, Mark. **Object models – strategies, patterns, and applications**. New Jersey: Object international, 1995.

CUNHA, Roberto Silvino. **Ferramenta para a geração de protótipos de sistemas baseado nas técnicas de pattern e framework através do repositório da ferramenta CASE System Architect**. 2001. 97 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.



FOWLER, Martin, SCOTT, Kendall. **UML essencial** Um breve guia para linguagem-padrão de modelagem de objetos. Porto Alegre: Bookman, 2ed, 2001.

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

GAMMA, Erich et al. **Design patterns** – elements of reusable object-oriented software. Massachusetts: Addison-Wesley, 1995.

JACOBSON, I. et al. **Software Reuse** - architecture process and organization for business success. New York: Addison-Wesley, 1997.

JACOBSEN, H. -Arno, KRÄMER, Bernd J. **A design pattern based approach to generation synchronization adaptors from annotated IDL**. New York, jan. 2001. Disponível em: <<http://www.eecg.toronto.edu/~jacobsen/abstracts/ase-abs.html>>. Acesso em: 29 mai. 2001.

INPRISE CORPORATION INC. **Visibroker** – CORBA technology from Borland. Disponível em <<http://www.inprise.com/visibroker>>. Acesso em: 23 ago. 2001.

LAURENT, Simon St., CERAMI, Ethan. **Building XML applications**. New York: McGraw-Hill, 1999.

LIMA Jr, Luiz Augusto de Paula. Objetos Distribuídos. In: IX Escola de Informática da SBC-Sul, 1., 2001, São José. **Anais...** Porto Alegre: UFRGS, 2001. p. 145-173.

MARCHAL, Benoit. **XML conceitos e aplicações**. São Paulo: Berkeley, 2000.

MOWBRAY, Thomas J., MALVEAU, Raphael C. **CORBA Design Patterns**. New York: Willey, 1997.

PAGE-JONE, Meilir. **Fundamentos do desenho orientado a objeto com UML**. São Paulo: Makron Books, 2001.

PHAMPSHIRE, Paulo. **O futuro do desenvolvimento de software no Brasil**. Disponível em: <[http://www.borland.com.br/artigos/artigo\\_phampshire\\_2.htm](http://www.borland.com.br/artigos/artigo_phampshire_2.htm)>. São Paulo, ago. 2001. Acesso em: 15 set. 2001.

PITTS-MOULTIS, Natanya, KIRK, Cheryl. **XML black book** – solução e poder. Tradução: Ariovaldo Griesi. São Paulo: Makron Books, 2000.

PREE, Wolfgang. **Design patterns for object-oriented software development**. New York: ACM, 1995.

SCHNEIDE, Ricardo Luiz. **Design patterns**, Rio de Janeiro, maio 1999. Disponível em: <[http://www.dcc.ufrj.br/~schneide/PSI\\_981/gp\\_6/design\\_patterns.htm](http://www.dcc.ufrj.br/~schneide/PSI_981/gp_6/design_patterns.htm)>. Rio de Janeiro, nov. 1999. Acesso em: 30 mai 2001.

SIEGEL, Jon. **CORBA** – fundamentals and programing. New York: John Willey, 1996.

TALIGENT. **Leveraging object-oriented frameworks**. Taligent Inc. white paper. Disponível em :<<http://www.taligent.com>>. Acesso em 01/08/2001.

SILVA, Roberto Silveira Silva. **Uma arquitetura baseada em CORBA pra Workflow de larga escala**, São Paulo, ago. 2000. Disponível em :<<http://www.ics.uci.edu/~rsilvafi/tese/>>. Acesso em: 10 ago. 2001.

## ANEXO 1 – IMPLEMENTAÇÃO DE UM OBJETO DE NEGÓCIO GERADO

```

unit Pessoa_impl;

interface

uses
  SysUtils,
  PersistentLayer_i, PersistentLayer_c,

  CORBA,
  Pessoa_i,
  Pessoa_c;

type
  TPessoa_callback = class;
  TPessoa = class;

  TPessoa_callback = class(TInterfacedObject, Pessoa_i.Pessoa_callback)
  protected
    {*****}
    {*** User variables go here ***}
    {*****}
  public
    constructor Create;
    procedure getInfo_callback ( const sucesso : Boolean;
                                const retorno : AnsiString);
  end;

  TPessoa = class(TInterfacedObject, Pessoa_i.Pessoa)
  protected
    {*****}
    {*** User variables go here ***}
    {*****}
    _tb : CORBAObject;
    _tableName : AnsiString;
    function _get_tb : CORBAObject;
    procedure _set_tb ( const tb : CORBAObject);
    function _get_tableName : AnsiString;
  public
    constructor Create;
    function insertDB ( const colName : Pessoa_i.strArray;
                       const colValues : Pessoa_i.strArray): Boolean;
    function updateDB ( const colName : Pessoa_i.strArray;
                       const colValues : Pessoa_i.strArray): Boolean;
    function delete ( const where : Pessoa_i.strArray): Boolean;
    function getAll : AnsiString;
    procedure getInfo ( const codigo : Integer;
                       const cliente : Pessoa_i.Pessoa_callback);
    property tb : CORBAObject read _get_tb write _set_tb;
  end;

```

```

    property tableName : AnsiString read _get_tableName;
end;

implementation

constructor TPessoa_callback.Create;
begin
    inherited;
end;

procedure TPessoa_callback.getInfo_callback ( const sucesso : Boolean;
                                             const retorno : AnsiString);
begin
end;

constructor TPessoa.Create;
begin
    inherited;
    _tableName:= 'Pessoa';
end;

function TPessoa._get_tb : CORBAObject;
begin
    Result := _tb;
end;

procedure TPessoa._set_tb ( const tb : CORBAObject);
begin
    _tb := tb;
    TableManager(_tb):= TTableManagerHelper.bind;
end;

function TPessoa._get_tableName : AnsiString;
begin
    Result := _tableName;
end;

function TPessoa.insertDB ( const colName : Pessoa_i.strArray;
                           const colValues : Pessoa_i.strArray): Boolean;
begin
    result:=TableManager(_tb).insert(persistentLayer_i.strArray(colName),
    persistentLayer_i.strArray(colValues),_tableName) = 0;
end;

function TPessoa.updateDB ( const colName : Pessoa_i.strArray;
                           const colValues : Pessoa_i.strArray): Boolean;
begin
    result:= TableManager(_tb).updateDB('UPDATE Pessoa SET codigo=' +
ColValues[0] + ',nome=''' + ColValues[1] + ''',fone=''' + ColValues[2] +
''',email=''' + ColValues[3] + ''' where codigo=' + ColValues[0])=0;
end;

function TPessoa.delete ( const where : Pessoa_i.strArray): Boolean;
begin

```

```
        result:=TableManager(_tb).DeleteDB('DELETE FROM Pessoa WHERE ' +
where[0])=0;
end;

function TPessoa.getAll : AnsiString;
begin
    result := TableManager(_tb).select('select * from Pessoa');
end;

procedure TPessoa.getInfo ( const codigo : Integer;
                             const cliente : Pessoa_i.Pessoa_callback);
begin
    cliente.getInfo_callback(true, 'r');
end;

initialization

end.
```

## ANEXO 2 – PROGRAMA PRINCIPAL DA APLICAÇÃO CLIENTE

```

program cliente;
{$APPTYPE GUI}
uses
  Forms,
  ComApp,
  uPrincipalCliente in 'uPrincipalCliente.pas' {FrmCliente},
  { interfaces }
  uPessoa in 'uPessoa.pas' {Pessoa: TWebPageModule} {*.html},
  uPessoaForm in 'uPessoaForm.pas' {PessoaForm: TWebPageModule} {*.html},
  Pessoa_c in 'c:\temp\Pessoa_c.pas',
  Pessoa_i in 'c:\temp\Pessoa_i.pas',
  uFinanceiro in 'uFinanceiro.pas' {Financeiro: TWebPageModule} {*.html},
  uFinanceiroForm in 'uFinanceiroForm.pas' {FinanceiroForm: TWebPageModule}
{*.html},
  Financeiro_c in 'c:\temp\Financeiro_c.pas',
  Financeiro_i in 'c:\temp\Financeiro_i.pas',
  uContas in 'uContas.pas' {Contas: TWebPageModule} {*.html},
  uContasForm in 'uContasForm.pas' {ContasForm: TWebPageModule} {*.html},
  Contas_c in 'c:\temp\Contas_c.pas',
  Contas_i in 'c:\temp\Contas_i.pas',
  uFornecedor in 'uFornecedor.pas' {Fornecedor: TWebPageModule} {*.html},
  uFornecedorForm in 'uFornecedorForm.pas' {FornecedorForm: TWebPageModule}
{*.html},
  Fornecedor_c in 'c:\temp\Fornecedor_c.pas',
  Fornecedor_i in 'c:\temp\Fornecedor_i.pas',
  uCidade in 'uCidade.pas' {Cidade: TWebPageModule} {*.html},
  uCidadeForm in 'uCidadeForm.pas' {CidadeForm: TWebPageModule} {*.html},
  Cidade_c in 'c:\temp\Cidade_c.pas',
  Cidade_i in 'c:\temp\Cidade_i.pas',

  { página principal }

  uHome in 'uHome.pas' {Home: TWebAppPageModule} {*.html};

begin
  Application.Initialize;
  Application.CreateForm(TFrmCliente, FrmCliente);
  Application.Run;
end.

```

## ANEXO 3 – UNIT DO PROGRAMA CLIENTE

```

unit uPessoa;
interface
uses
  Windows, Messages, SysUtils, Classes, HTTPApp, WebModu, HTTPProd,
  CompProd, PagItems, SiteProd, DB, DBClient, WebAdapt, WebComp, DBAdapt,
  WebForm, MidItems, CORBA,

  Pessoa_c, Pessoa_i;

type
  TPessoa = class(TWebPageModule)
    AdapterPageProducer: TAdapterPageProducer;
    AdpFrmPage: TAdapterForm;
    AdpGrid: TAdapterGrid;
    DataSetAdp: TDataSetAdapter;
    cds: TClientDataSet;
    AdapterCommand: TAdapterCommandGroup;
    CmdEditRow: TAdapterActionButton;
    CmdNewRow: TAdapterActionButton;
    CmdCancel: TAdapterActionButton;
    procedure WebPageModuleCreate(Sender: TObject);
    procedure cdsBeforePost(DataSet: TDataSet);
  private
    { Private declarations }
  protected
    fPessoa: Pessoa;

    procedure bind_object;
    procedure getXML;
  public
    { Public declarations }
  end;

  function Pessoa: TPessoa;

implementation

{$R *.dfm} {*.html}

uses WebReq, WebCntxt, WebFact, Variants, uPrincipalCliente;

function Pessoa: TPessoa;
begin
  Result := TPessoa(WebContext.FindModuleClass(TPessoa));
end;

procedure TPessoa.bind_object;
begin
  fPessoa:= TPessoaHelper.Bind();
  getXML;
end;

procedure TPessoa.getXML;

```

```

var
  rtn: WideString;
  str: TStringStream;
begin
  rtn := fPessoa.getAll;
  //pega a XML de uma stream
  str := TStringStream.Create(rtn);
  try
    cds.Close;
    cds.LoadFromStream(str);
    cds.Open;
  finally
    //seta a chave primária
    cds.Fields[0].ProviderFlags := [pfInKey];
    str.Free;
  end;
end;

procedure TPessoa.WebPageModuleCreate(Sender: TObject);
begin
  bind_object;
  CmdEditRow.PageName := 'PessoaForm';
  CmdNewRow.PageName := 'PessoaForm';
end;

procedure TPessoa.cdsBeforePost(DataSet: TDataSet);
var
  colName: strArray;
  colValues: strArray;

  procedure carrega;
  var
    i: integer;
  begin
    setLength(colName, cds.Fields.count);
    setLength(colValues, cds.Fields.count);
    for i:=0 to cds.Fields.count-1 do
      begin
        colName[i] := cds.Fields.Fields[i].Name;
        colValues[i] := cds.Fields.Fields[i].AsString;
      end;
    end;
  begin
    if cds.State = dsInsert then
      begin
        carrega;
        fPessoa.insertDB(colName, colValues)
      end
    else
      if cds.State = dsEdit then
        begin
          carrega;
          fPessoa.updateDB(colName, colValues);
        end;
      end;
  end;
end;

```



```
initialization
  if WebRequestHandler <> nil then

WebRequestHandler.AddWebModuleFactory(TWebPageModuleFactory.Create(TPessoa,
TWebPageInfo.Create([wpPublished      {,      wpLoginRequired}],      '.html'),
crOnDemand, caCache));

end.
```